FLORIDA INTERNATIONAL UNIVERSITY

Miami, Florida

A REALTIME INTERNET BASED QUALITY CONTROL APPLICATION FOR HURRICANE SURFACE WIND OBSERVATIONS

A thesis submitted in partial satisfaction of the requirements for the degree of

MASTER OF SCIENCE

IN

COMPUTER SCIENCE

by

Luis R. Amat, Jr.

1998

To: Dean Arthur W. Herriott
College of Arts and Sciences

This thesis, written by Luis R. Amat, Jr., and entitled A Realtime Internet Based Quality Control Application for Hurricane Surface Wind Observations, having been approved in respect to style and intellectual content, is referred to you for your judgment.

We have read this thesis and recommend that it be approved.

Dr. Yi Deng

Dr. Mark Powell

Dr. Raimund Ege, Major Professor

Date of Defense: November 20, 1998

The thesis of Luis R. Amat, Jr. is approved

Dean Arthur W. Herriott
College of Arts and Sciences

Dean Richard L. Campbell
Division of Graduate Studies

Florida International University, 1998

DEDICATION

I dedicate this thesis to my Mother and Father, Elaine and Luis. Their support for me has always been unconditional and they have always put their children’s needs before theirs. Their love and support has enabled me to reach my goals throughout my life. Thank you, Mom and Dad.

ACKNOWLEDGMENTS

I would like to thank the following people for their unwavering support, patience and expertise: The members of my committee (Dr. Raimund Ege, Dr. Mark Powell and Dr. Yi Deng), my family (Mom, Dad, Mauri and all my grandparents), my colleagues and supervisors at AOML (Sam Houston, Sonia Otero, Nirva Morisseau-Leroy, Nicholas Carrasco, Shirley Murillo and Jason Dunion), all my friends at AOML, my friends elsewhere, my co-members on the original WANDA development team (Cary Bakker and Philippe Dubosq), and the “Man Upstairs”. Each individual in this group helped to shape this project and my efforts on this project in their own way. You have all contributed more than you will ever know to my education and growth as a professional and as an individual. Thank you all for letting me learn from you.

ABSTRACT OF THE THESIS

A REALTIME INTERNET BASED QUALITY CONTROL APPLICATION FOR HURRICANE SURFACE WIND OBSERVATIONS

by

Luis R. Amat, Jr.

Florida International University, 1998

Miami, Florida

Professor Raimund K. Ege, Major Professor

This thesis chronicles the design and implementation of a Internet/Intranet and database based application for the quality control of hurricane surface wind observations. A quality control session consists of selecting desired observation types to be viewed and determining a storm track based time window for viewing the data. All observations of the selected types are then plotted in a storm relative view for the chosen time window and geography is positioned for the storm-center time about which an objective analysis can be performed. Users then make decisions about data validity through visual nearest-neighbor comparison and inspection. The project employed an Object Oriented iterative development method from beginning to end and its implementation primarily features the Java programming language.

Table of Contents

1. **Introduction**

Over the past ten years tropical cyclones have caused losses averaging roughly $5 billion per year in the United States alone (Powell, 1998). Accurate forecasts of hurricane landfalls can help those living in threatened communities to prepare their homes and property and leave areas subject to flooding, but significant damage is, however, inevitable. The effectiveness of the response to such disasters depends on the accurate and timely acquisition and interpretation of information regarding the magnitude and geographical distribution of the damage. This information helps emergency managers and others determine which communities were most devastated and which require immediate attention; it also assists decision-making associated with the recovery process. "Rapid assessment of tropical cyclone disasters requires physical measurements of the quantity forcing the damage. For hurricanes, the wind provides the forcing which builds waves and storm surge, buffets houses, and uproots trees" (Powell, 1998).

Tropical cyclones are monitored globally by space, aircraft, land and marine based observing systems. Advances in computing and communications have made it possible to obtain these observations in near real-time. However, scientists involved in operational forecasting and basic and applied research on hurricanes have few tools that enable real-time interaction with, and analysis of, observations gathered in tropical cyclones. Hurricane wind fields are currently determined subjectively based on a specialist's interpretation of flight-level reconnaissance data, satellite observations, pressure-wind relationships and available surface data. Furthermore, there is currently no operational objective method for assimilating and synthesizing disparate observations into a consistent wind field.

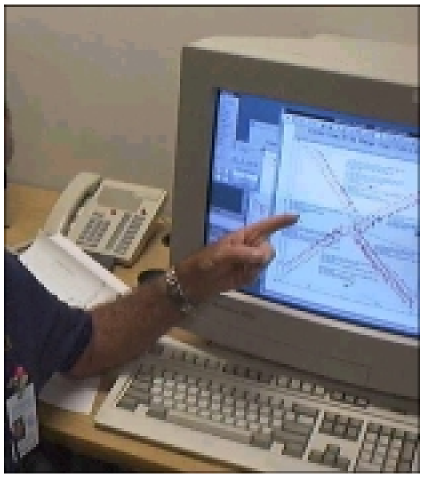
The National Oceanic and Atmospheric Administration's (NOAA) Hurricane Research Division (HRD) has been producing real time analyses of tropical cyclone surface wind observations on an experimental basis since 1993 (Burpee, 1994). HRD's analyses, as described by Powell, et. al. (Powell, 1998), are created by compiling all available observations relative to the storm center about which the analysis is made. Some of these observations are taken from Air Force and NOAA aircraft, ships, buoys, Coastal Marine Automated Network (CMAN) platforms and surface airways (airports). Before going through a quality control session the observations are adjusted to simulate the same observations at a common height (10 meters), exposure (marine or land), and averaging period (maximum sustained 1 minute wind speed). Because several hours of observations are usually required to provide sufficient data for an analysis the resulting objective analysis represents the mean state of the storm during the chosen time period. A typical 10 hour reconnaissance mission will yield two to three analyses.

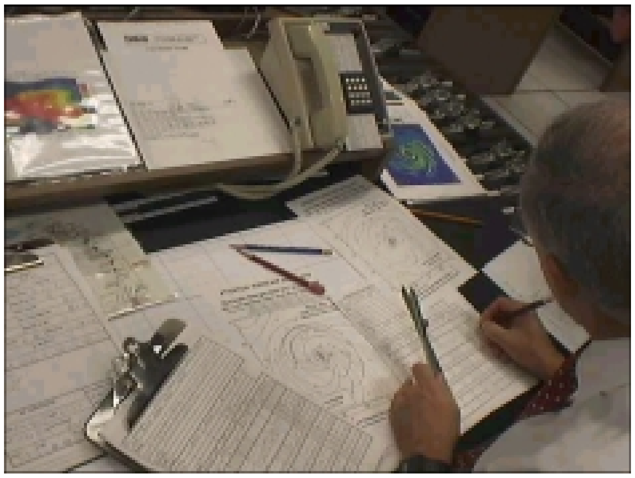
The primary product of an analysis, a streamline and isotach contour plot, is designed to aid meteorologists in forecasting a storm's behavior by helping them determine the storm's current intensity and the extent of its damaging winds. During a storm's landfall, an analysis image coupled with the affected geography can help determine which areas are suffering from the most violent winds and storm surge. These tools should supply emergency managers with enough information to help limit confusion and maximize the efficiency of early search, rescue and recovery efforts in these hard hit areas. Many research and commercially oriented groups have also expressed interest in obtaining access to hurricane wind field data in flat file and graphical format.

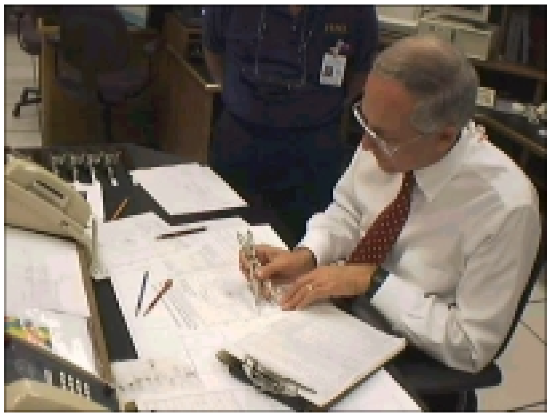
2. **H*WIND: Existing System**

2.1 *Existing System Overview*

HRD’s Hurricane Wind Analysis System (H*WIND), whose initial implementation, Wanda (“Wind Analysis Distributed Application”) (Amat, Jr., 1998) saw extended use during the 1996, 1997 and 1998 hurricane seasons, is an Object Oriented (OO), workstation based, software product that partially automates the wind analysis process discussed above (Powell, 1998). Currently, the application fetches data, wind observations and a storm track, from a repository of flat files. The repository is populated through several cron activated UNIX scripts that retrieve near real time data from the National Center for Environmental Prediction (NCEP) via the National Hurricane Center (NHC) at the Florida International University (FIU) campus during a storm via HRD’s dedicated network connection. The data are then processed, quality controlled, passed on to the analysis server, and then displayed on a workstation screen as graphical products that are sent as hard copies to NHC’s hurricane specialists or other clients.

a. 

b. 

c. 

d.

Figure 1. H*WIND's current version (WANDA) at use in real-time at the National Hurricane Center during Hurricane Georges in September, 1998. Figure 1a. shows a plot of Air Force reconnaissance data using the Quality Control client. Figures 1b, c, and d show Hurricane Specialist Max Mayfield using a surface wind contour and streamline plot, H*WIND's primary product, to help in making an advisory.

Three subsystems are employed to automate an analysis: 1) Quality Control, 2) Analysis Automation and 3) Output Generation. A subsystem, in OO terms, groups tightly coupled classes of objects, such as those whose object instances frequently interact with each other, together so that the classes can be viewed as a single entity (Yourdon, 1994). Because H*WIND can be naturally divided into these subsystems, it is easy to examine each separately.

2.1.1 *Quality Control*

In the Quality Control subsystem, the goal is to arrive at a plot window containing observations, a storm track and geography so that decisions can be made about the validity of the potential analysis data set (Figure 2). A user may choose a synoptic (earth relative) or storm relative view of the data. The synoptic view helps the user study large scale weather systems at a particular time; it does not include a storm track (Figure 3). The storm relative view converts observations to a range and bearing relative to the storm over a period of time. Geography is then positioned relative to the storm at a chosen time. The storm relative view helps fill areas that are devoid of data by including all data during a period of time when the storm is considered to be near steady state. See Figure 4 for a comparison of synoptic and storm relative coordinates. In the plot window, observations are represented as are graphic representations of wind speed and direction called wind barbs (see Figure 5). The user is provided with zooming, distance and data inspection tools to facilitate the task of selecting an acceptable set of data from the plot window.

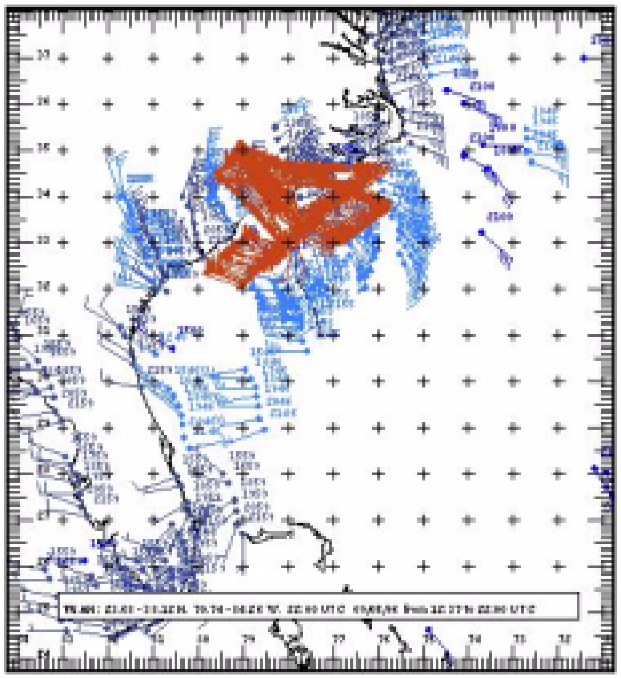
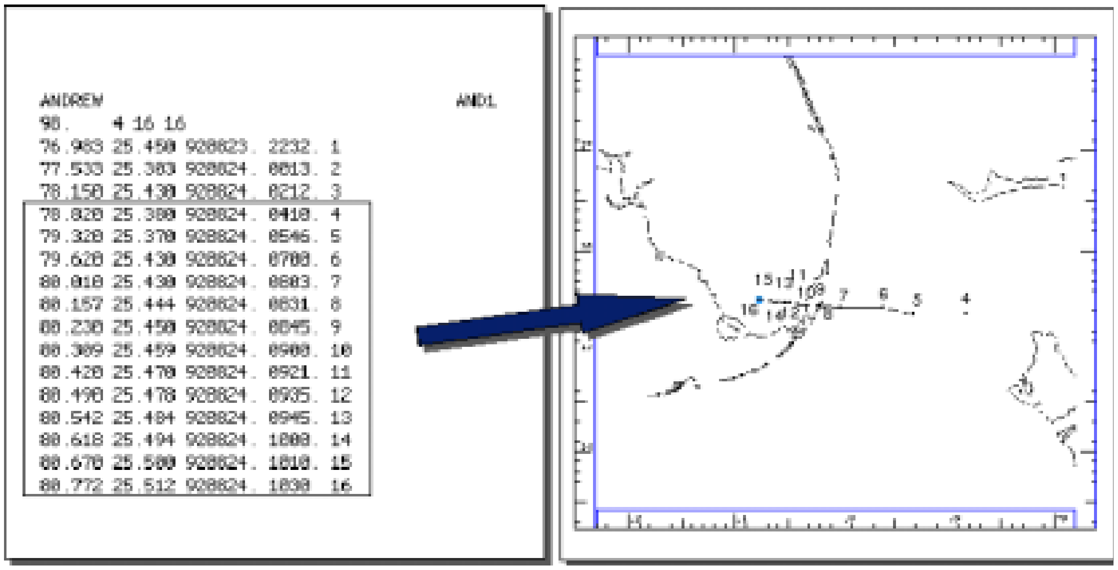


Figure 2. Quality Control Plot Window. Storm relative view of Hurricane Fran near landfall at North Carolina in 1996. Figures accompanying the wind barbs indicate the times (UTC) of the observations.



```
ANDREW                                        AND1
90.     4 16 16
76.903 25.450 920823. 2232. 1
77.533 25.383 920824. 0013. 2
78.150 25.430 920824. 0212. 3
78.820 25.300 920824. 0410. 4
79.320 25.370 920824. 0546. 5
79.620 25.430 920824. 0700. 6
80.010 25.430 920824. 0803. 7
80.157 25.444 920824. 0831. 8
80.230 25.450 920824. 0845. 9
80.309 25.459 920824. 0900. 10
80.420 25.470 920824. 0921. 11
80.490 25.478 920824. 0935. 12
80.542 25.484 920824. 0945. 13
80.618 25.494 920824. 1000. 14
80.670 25.500 920824. 1010. 15
80.772 25.512 920824. 1030  16
```

Figure 3. Partial Storm Track for Hurricane Andrew in 1992. Each row in the figure on the left indicates a Storm Track Fix. The column fields on the left of the figure are as follows: longitude (degrees West), latitude, date, UTC time and an enumeration of the storm track fixes. The box indicates which fixes are plotted on the right.

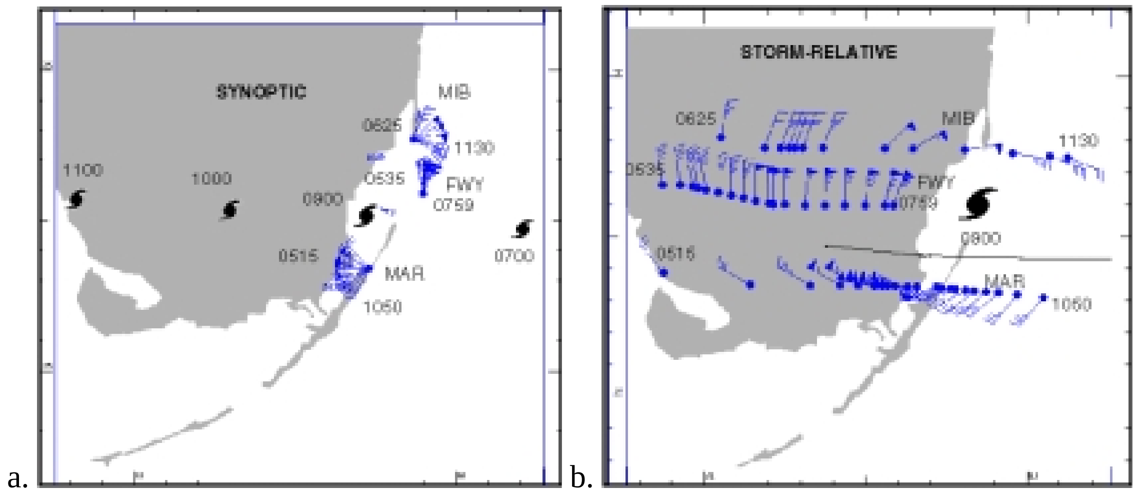


Figure 4. Comparisons of Synoptic (a) and Storm Relative (b) views of Hurricane Andrew observations at Fowey Rocks, Miami Beach and the yacht "Mara Cu" (Powell, 1998).

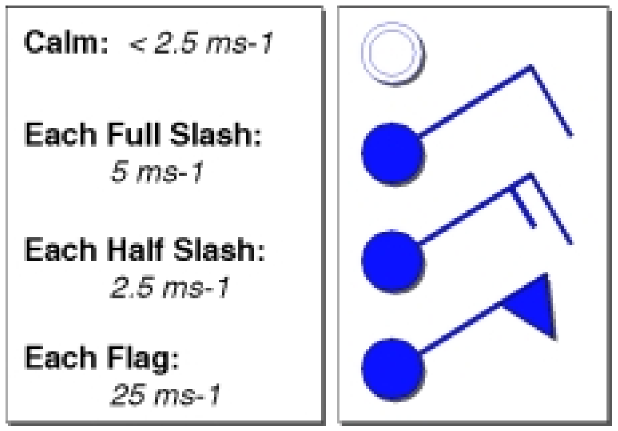


Figure 5. Wind Barbs: Orientation shows wind direction and end of barb shows speed. The last three barbs indicate winds coming from the northeast.

2.1.2 *Automated Analysis*

Once the data go through quality control, they are passed, along with a storm track, through a series of Analysis Automation subsystem components. Each component is distributed over two machines, a NextStep client and a VAX/VMS server where the analysis programs reside. H*WIND uses state machines to orchestrate all of the transitions involved in analysis automation. Included in this automation is the archival of all steps of an analysis for research purposes. Through these archives, any analysis can be traced back to its components' results and intermediate data sets. The state machines also report errors and handle interruptions due to errors by popping states off a "state stack" once a failure is encountered. This way a component can only be completed and move to the next component if no errors are encountered. The use of state machines also makes maintenance much easier in that errors are accurately reported and changes in functionality often only involve changing the machine to address the new needs.

2.1.3 *Output Generation*

The Output Generation subsystem creates the graphical representation of the wind fields as discussed earlier (Figure 6). This subsystem relies on an in house graphics package to display an analysis result (streamlines and isotach contours) on the client workstation screen where a meteorologist can then annotate it and save it to an encapsulated postscript document. The document can then be delivered to NHC forecasters for use as guidance in preparing hurricane and tropical storm warnings and advisories. Analyses are also archived for research purposes and for distribution on the World Wide Web. For example, an archived analysis may be accessed, sampled at some grid interval and written to a specific GIS (Geographic Information System) file format. A specific area's emergency management team can then overlay these data over their local geography and make damage assessments or emergency decisions regarding the local population base.

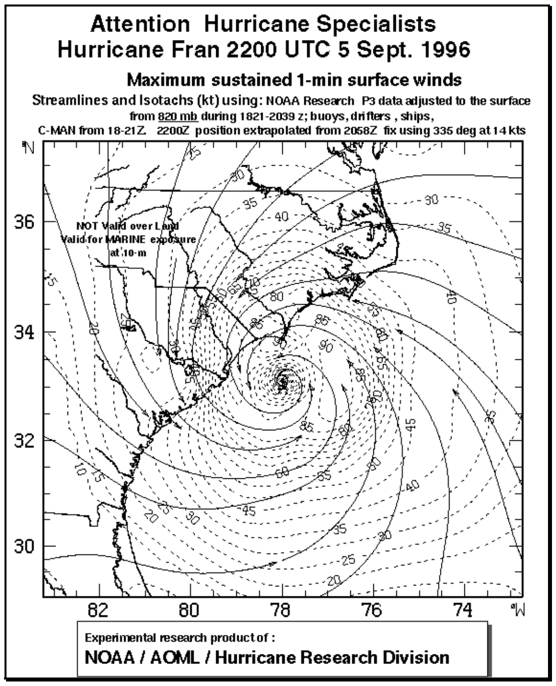

Attention Hurricane Specialists
Hurricane Fran 2200 UTC 5 Sept. 1996
Maximum sustained 1-min surface winds
Streamlines and Isotachs (kt) using: NOAA Research P3 data adjusted to the surface from 820 mb during 1821-2039 z; buoys, drifters, ships, C-MAN from 18-21Z. 2200Z position extrapolated from 2058Z fix using 335 deg at 14 kts
NOT Valid over Land
Valid for MARINE exposure at 10 m
°N
36
34
32
30
82
80
78
76
74
°W
Experimental research product of :
NOAA / AOML / Hurricane Research Division

Figure 6. Primary analysis product. Streamlines and isotach contour plot for Hurricane Fran in 1996.

2.2 *Limitations of the Existing System*

Although very useful, the current system has some inherent limitations. Firstly, the use of flat file data archives marries the application to a specific filesystem and, at the same time, raises data integrity and security issues. For example, under UNIX, a developer must take measures to set the correct file protection modes on any archived data, and a separate group for H*WIND's users is necessary for minimal archive data security. Secondly, lack of portability, an inherent problem in most compiled applications, is amplified in this case through the use of code specific to the NextStep environment. A third limitation lies in the method of distribution of the analysis processes. A more robust method than using ftp, rsh, rcp and VMS DCL script can and should be used.

3. **H*WIND: Improved System**

3.1 *Improved System Overview and Related Work*

A reworked H*WIND product will manifest substantial improvements during the 1999 Atlantic Hurricane Season. Although this thesis primarily involves H*WIND's Quality Control subsystem, concurrent improvements to other subsystems are also discussed because of their relevance to the entire project and because the three subsystems are, in many ways, inherently similar.

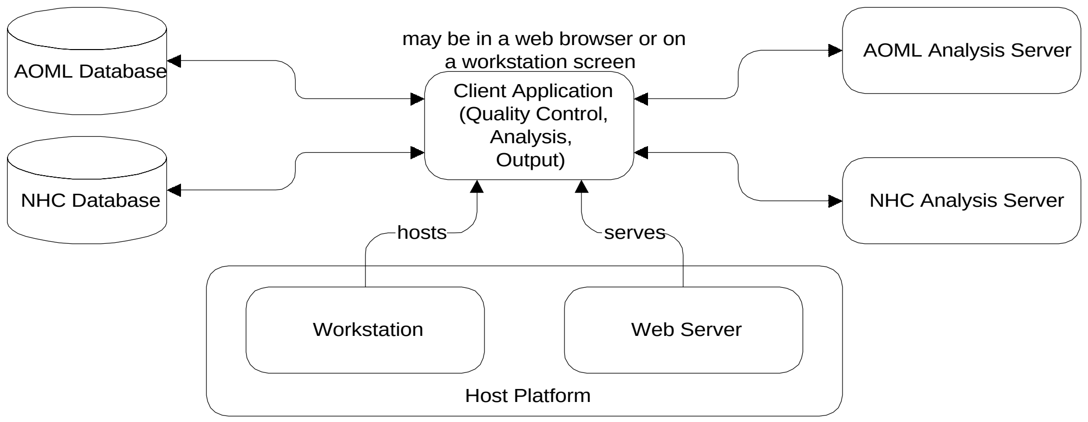


Figure 7. A generalized view of the proposed H*WIND project. Redundant databases and analysis servers will ultimately be used and the application will run on both the web and on client workstations.

3.2 *Overall System Design Strategies*

3.2.1 *Object-Oriented Approach*

Modeling the real world through the object oriented (OO) paradigm has been a leading trend in the last decade of software development. Through the use of abstract data types and data encapsulation within classes of objects, objects represented in software send and receive messages to each other that are meaningful to them via "publicly visible" interfaces. Developing software in this manner helps programmers and designers alike by allowing the creation of intricately dynamic software with potentially reusable and modular code. Furthermore, the OO paradigm extends beyond the scope of the programming language to pervade the overall design philosophy. Object Orientation benefits entire projects, not just individual subsystems, through code reuse, the use of design patterns, rapid-development, more accurate estimation and more effective and efficient testing and maintenance. In light of these benefits and because vocabulary, notation and strategies are easily shared throughout an object oriented project, this project and its subprojects use an OO iterative development method from beginning to end. The OO manager's strategy as described by Edward Yourdon "is quite simple: develop an initial model as best you can, but then be prepared to revise that estimate with the completion of each version/iteration of the system" (Yourdon, 1994).

In general, we are developing an object oriented framework designed to allow the construction of reusable objects that combine scientific logic with persistent data storage. More specifically, we employ a highly portable object oriented programming language (Java), object relational database technology and object distribution to offer a distributed, dynamic, interactive and platform independent product.

3.2.2 *Software Reuse: The HRD Packages*

When starting the design of the system, I took the view that much of the specialized functionality to be implemented should not be bound to this specific project, so code and design reuse became priorities. We needed packages that fit HRD's areas of expertise: meteorology and mapping. “The promise of easy reuse of classes, “ writes Brooks, “with easy customization via inheritance, is one of the strongest attractions of object oriented techniques” (Brooks, Jr., 1995). Schach suggests that one of the potential advantages of deliberate reuse “is that components specially constructed for use in future products are more likely to be easy and safe to reuse; such components are generally well documented and thoroughly tested” (Schach, 1996). The HRD packages grew out of a need for cohesive units of specialized logic and functionality that could still be extended to fit the needs of individual applications. The current upper level packages, as shown in Figure 8, include "hrd.geography" (classes representing geographical objects such as global positions, global areas, maps, etc.), "hrd.meteorology" (classes representing meteorological objects such as wind observations, storm track fixes, storm tracks, etc.), "hrd.math" (classes representing mathematical objects such as vectors, matrices and angles), "hrd.apps" (classes and subpackages pertaining to applications and the various subsystems of the project), “hrd.db” (classes and interfaces for interfacing with a database), “hrd.awt” (classes and interfaces for creating graphical components and interfacing with the user), and the utility packages "hrd.util" (non-graphical utilities) and "hrd.file" (abstract filesystem manipulation utilities). During my design of the specialized H*WIND application components (those classes which are specialized for use exclusively by the QCClient), I treated the rest of the hrd packages’ classes as if they were pre-existing software components, “unit[s] of packaging, distribution, and maintenance” (Orfali, 1996). A component assembly approach helped me to improve those base level components in such a way that they would be general enough and decoupled enough from each other to allow for the safe and rapid development of future applications. The individual hrd packages and classes are presented in greater detail in Appendix A.

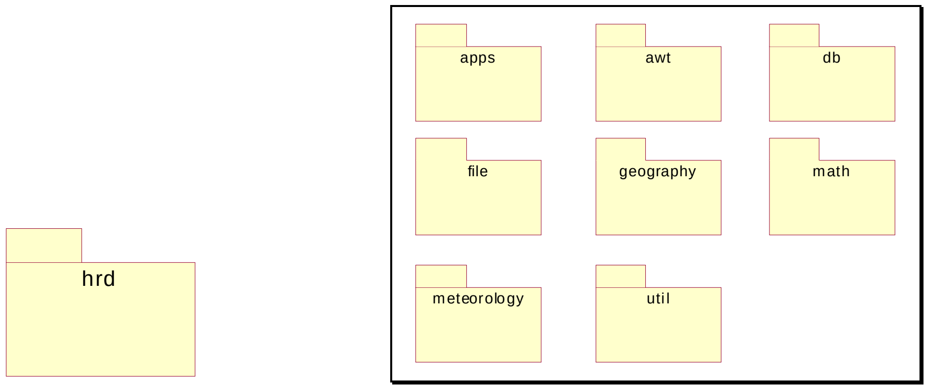


Figure 8. The HRD package hierarchy. The “hrd” package is the root of the hierarchy. All packages on the right are contained within the “hrd” package.

3.2.1 *Migrating to A Database Based Solution*

All aspects of the project should rely on a central database for most, if not all, data storage and delivery. The database approach to data archival provides several advantages over traditional file processing approaches. Among these are: 1) self description, a database’s ability to not only contain data but to contain a description of itself, 2) insulation between programs and data, 3) data abstraction, 4) support of multiple views of the same data and 5) sharing of data and multi-user transaction processing. Furthermore, a good database system automatically solves many of the problems inherent in most data processing projects such as controlling redundancy, restricting unauthorized access, providing for persistent storage of program objects and data structures, representing complex relationships among data, and enforcing integrity constraints for the data (Elmasri and Navathe, 1994). HRD investigated several off-the-shelf object-oriented and object-relational databases. Object-Oriented databases offer some advantages over relational databases; we will cite two: persistent object storage and inheritance. Edward Yourdon writes that the ideal solution for an OO system is an OO database because "every one of the 'persistent' classes and objects in the software would correspond exactly to a database object managed by the [OO database management system (OODBMS)]" (Yourdon, 1994). This feature is important because all of a program's rich types must be converted to the more primitive types found in a Relational DBMS. This impedance problem, as it is sometimes called, presents a problem because it creates strong coupling between the application and the DBMS (Jacobson, 1992).   The ability to use inheritance in an OO database is also an advantage in much the same way as it is to an OO application. It, above all, promotes the concepts of reuse and specialization. Object-relational databases combine both philosophies (OO and relational) and tend to represent not only the best qualities of OO databases, but also those of relational databases, namely a relational DBMS' superior storage algorithms.

A preliminary database schema for a subset of the data to be used already exists and is the subject of a Masters Thesis at Florida International University (Morisseau-Leroy, 1997). Two databases, one at AOML (NOAA's Atlantic Oceanographic and Meteorological Laboratory; HRD's home) and one at NHC, will eventually be used for redundancy in case of a landfall in the South Florida area or during periods of heavy use or inevitable outage.

3.2.4 *Platform Independence*

Platform independence and client side deployment on both workstations and the World Wide Web are integral parts of this project. H*WIND's code implementation features Sun Microsystems' Java programming language because its "write once, run anywhere" strategy lends itself to the simultaneous development and maintenance of workstation and web versions. With Java, the H*WIND application, or suite of applications, needs to be written only once and run either as a Java application when running the workstation version or as a Java applet embedded in HTML for the web version. Platform independent code coupled with web deployment allows users to not only use any machine they want (as long as it supports the Java virtual machine), but also to be wherever they want on the Internet.

3.2.5 *Distributed Objects*

For back end computational load sharing, we are using a distributed OO system over several distinct application, analysis, and data servers. Distributed objects are becoming more common in software applications as a way to distribute processing components over heterogeneous and geographically distant computing platforms. A distributed object is essentially a component, writes Orfali, "a blob of self contained intelligence that can interoperate across operating systems, networks, languages, applications, tools, and multivendor hardware" (Orfali, 1996). This allows a real time or operational system to be more robust because of the redundancy of computational nodes (should one machine become unavailable another one can replace it) as well as more efficient because each platform is used to perform tasks best suited to its hardware. Distributed object (DO) technology will provide an immediate solution in the area of analysis automation. Because of the time needed to port the analysis components to an OO programming language (OOPL), the current surface wind analysis programs, written in, FORTRAN 77, are wrapped in Java objects and distributed such that each wrapped object acts as an agent for the scientific code that it contains and has access to the other objects in the system. HRD has investigated several DO options and is siding with a product that supports CORBA (Common Object Request Broker Architecture), IDL (Interface Definition Language) and IIOP (Internet Inter-Orb Protocol) for an industry standard system that can be distributed to clients over the Internet. We may also employ Java's Remote Method Invocation (RMI) technology instead of or in addition to a CORBA ORB to take advantage of our heterogeneous client/server environment (Java on both the client and the analysis server sides). Object distribution in a subset of the project is also the subject of a Masters Thesis at Florida International University (Otero, in progress).

4. **Analysis and Design**

With OO methodologies, the boundary between analysis and design, or the what and the how, are blurred and indistinct. "[T]his [ambiguity] may have been accidental in the early days of OO," proposes Edward Yourdon, "but now it is a conscious and deliberate feature of the methodology…. This is particularly true … for prototyping environments, and where the software engineers have a common development environment for *all* their project activities" (Yourdon, 1994). In this section, I start with a set of requirements and use the notation and process of object-oriented analysis and design to reflect as closely as possible what lead me to an implementation.

4.1 Requirements

While a great deal of the concepts conveyed above in the section on the Improved System and Related Work permeate the entire project, this thesis focuses solely on the client side development and deployment of the Quality Control subsystem for the H*WIND project and its integration with a database. Sections 3.2.1 through 3.2.4 (*Object Oriented Approach*, *Software Reuse*, *Migrating to a Database Based Solution*, and *Platform Independence*) of the section describing the overall system design strategies best represent the overall concepts that influenced the development of the thesis. These conceptual improvements helped in focusing in on the Quality Control system's functional and dynamic requirements. They also provided the first hints as to the object hierarchy required to fulfill those requirements.

Preliminary meetings with HRD personnel coupled with my experience gained through the development of the initial version of H*WIND (formerly named WANDA: "Wind Analysis Distributed Application") helped produce a minimal list of product expectations that echo some of the descriptions given above in the section regarding the improved system. The Quality Control system should, in general, provide the following functionality with no precedence implied by the order:

1. Provide database connectivity to the subsystem. H*WIND should be treated as a “database-centric application”.
2. Plot atmospheric event (specifically tropical cyclone) data in synoptic (earth relative) and storm relative coordinates along with a storm track.
3. Provide a query mechanism for storm track selection.
4. Provide a query mechanism for measurement platform (e.g. specific aircraft, satellite or ground based platforms) and physical observation selection (e.g. temperature, wind speed, pressure, etc.).
5. Provide both real-time (operational) and post storm (research) modes for storm track and data selection. Using a database makes this task much easier than attempting the same task with only flat files.
6. Provide accurate geographical features (maps) behind the plot.
7. Provide a query mechanism for geographic region and geographic data selection.
8. Provide a query mechanism for individual datum inspection (including storm track fixes).
9. Provide a mechanism for data edition, creation and removal. This includes storm track fixes and atmospheric data, but not geographic data. The existing database schema (Morisseau-Leroy, 1997) and accompanying database side PL/SQL packages make provisions for these functions.
10. Provide zooming and distance calculation features for the plot and geography.

11. Provide a mechanism to alert the user of any new data that has arrived at the database during the session. While the user will be able to check for new data at any time, a separate thread or process continually running checks is not absolutely necessary, and may be considered a luxury.

Appropriate OO Analysis (OOA) and Design (OOD) documents as well as object interaction diagrams for all objects and object life history diagrams for selected objects were completed throughout different stages of the project and were consulted heavily throughout the project. The purposes of OO Analysis and Design documents are to capture the user requirements for a given system and to define the software system in a way that it will satisfy the requirements, respectively. The OOA served both as a way to efficiently review the requirements with the project supervisors and users and as a template leading to the construction of an OOD. Not only does the OOD document define the software system as mentioned above, but it also takes into account hardware and software constraints, interface design, testing methods, distribution and any other details that were most likely ignored in the OOA.

Although H*WIND's design employs UML (Unified Modeling Language) notation and terminology, I strayed from a strict UML method which calls for a use-case driven analysis similar to that described by Ivar Jacobsen in his OOSE (Object Oriented Software Engineering) method (Jacobsen, 1996). Such an analysis typically requires that object discovery occur after or during the design of a requirements document using use cases. At times, however, it is useful "to develop a logical view of the system using problem domain objects, that is, objects that have a direct counterpart in the application environment…" (Jacobsen, 1996). Prior to UML, James Rumbaugh supported this view by writing that "[t]he first step in analyzing the requirements is to construct an object model…. Information for the object model", he writes, " comes from the problem statement, expert knowledge, and general knowledge of the real world" (Rumbaugh, 1991). Although Jacobsen's OOSE method dictates that object discovery should typically occur after use case specification, he concedes that such a problem domain model will help the designer "develop a noun list which will be a strong support when [specifying] use cases." He goes on to support the idea of problem domain modeling in certain situations and writes that "a problem domain model… will serve as a very solid input to system development." In some projects where such a problem domain already exists, the "idea of the system to be developed [is] quite mature, and the development of the requirements model and the analysis [can] be done in a quite straightforward manner…" (Jacobsen, 1996). H*WIND is such a project. In it I chose to discover problem domain objects first and then follow with a use case analysis.

4.2 *QCClient Domain Classes*

The H*WIND QCClient application uses each of the classes in the HRD packages directly or through specialized subclasses. The classes contained within the HRD packages came about mainly because they were each necessary to the development of the QCClient. They directly or indirectly relate to the problem stated earlier in the introduction to thesis and to the requirements listed earlier in this section. As I discovered the classes that I would need for the client, I also found that I could define more general superclasses that would hold common or abstract attributes and services for several or potentially several classes. In other words, class discovery in the client did not necessarily come after class discovery in the HRD packages. Figure 9 contains the problem domain class hierarchy for the QCClient. Below I will briefly describe each important class or major superclass in the hierarchy and explain its role in the H*WIND application. Please note that not all of the classes involved in the client are listed below, only those which directly relate to the problem.

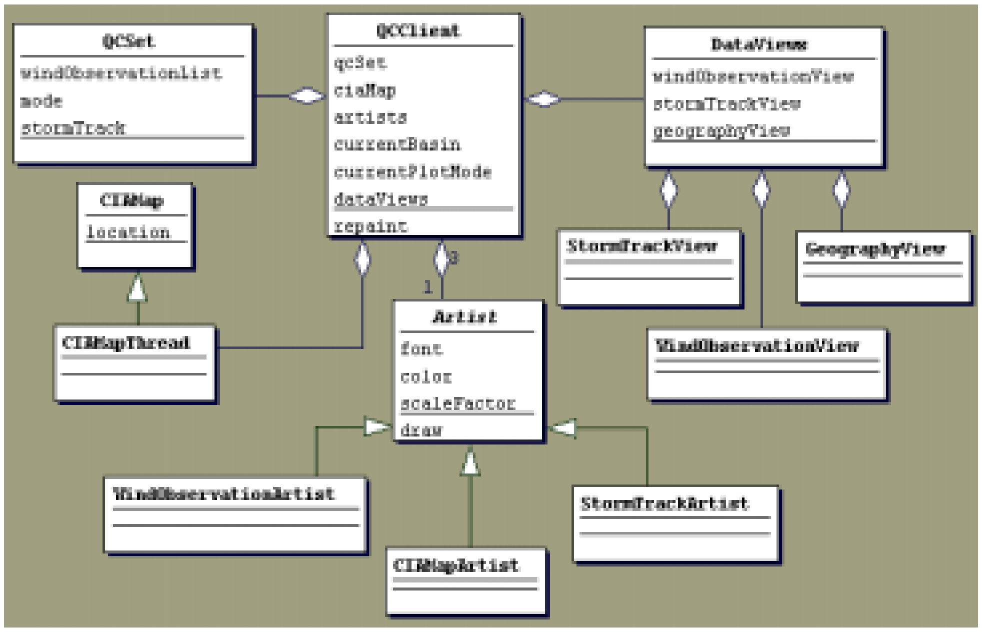


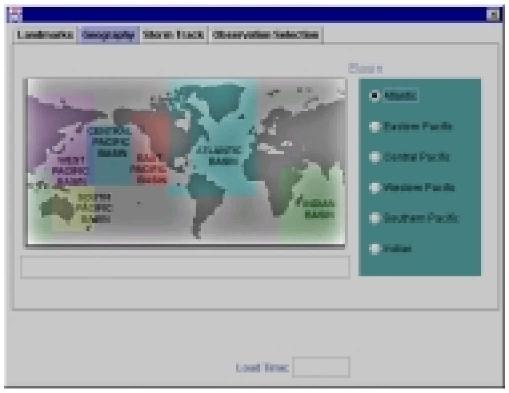
Figure 9. The QCClient Domain class hierarchy.

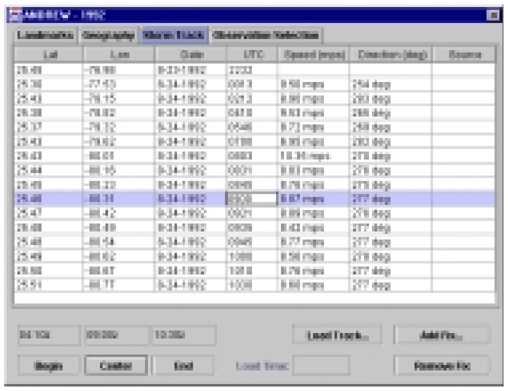
- <u>QCClient</u>: This is the primary class for the problem domain; it contains the *main* method. It is a subclass of the *java.awt.Canvas* class and it represents the drawable surface upon which observations, storm tracks and geography are plotted and is responsible for handling mouse events that occur within its bounds. It keeps track of the current state of the application (current tool selection, current basin selection [Atlantic, Eastern Pacific, etc.] and current zoom area) and is a container for the various objects that make up the application such as the Artists (delegates that draw the different components of a plot), the QCSet (the data being examined) and the CIAMap (the points representing the map for the current basin).

- QCSet: This class holds the current data that are being examined. These data include a list of *hrd.meteorology.WindObservation* instances, a hash table containing any edited WindObservations, a list of *QCSetErrors* for holding any reports generated by the experimental automated quality control system and, optionally, but most commonly, an *hrd.meteorology.StormTrack* object and a storm name. This class also keeps track of the current plot mode (synoptic or storm relative) and is responsible for fetching and inserting QCSets and their components from and into the H*WIND database and/or the filesystem.
- hrd.geography.CIAMap and CIAMapThread: These classes are parent and child and work together to load and store the various *hrd.geography.CIAMapPositions* that make up the geography for a specified storm basin. The available basins are as follows: Atlantic, Eastern Pacific, Central Pacific, Western Pacific, South Pacific and Indian. The relatively voluminous maps can either be stored in the database as large tables of positions or on the file system as serialized Java objects. All the points for a basin are held in memory instead of simply using an image because when a user zooms into different areas in the basin, he needs to see the geographical features present in that area. A zoomed image would show up as an aliased line with no

detailed features, but redrawing the points for a certain area instead yields a high quality representation of the geography. The specialized CIAMapThread subclass is used to load the map in a separate thread so that the user can continue working while waiting for the map to load.

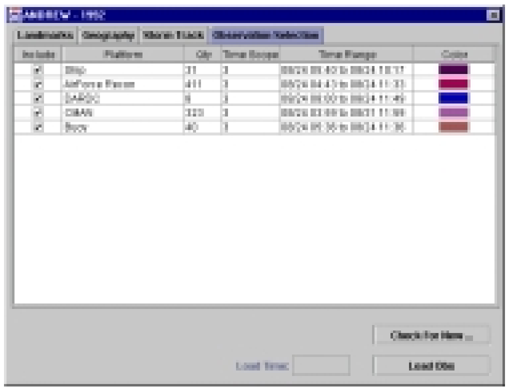
- Artist: The abstract superclass for each of the delegates responsible for drawing the different parts of a QCSet whenever the QCClient's *paint* method is invoked. Its subclasses include *CIAMapArtist*, *WindObservationArtist*, *StormTrackArtist* and *LandmarkSetArtist*. The Artist class is not designated as an interface because it is required to hold attributes common to each of the subclasses such as color, font, font color and scale factor in addition to defining a *draw* method for each subclass to implement.
- DataViews: A window containing detailed user views of the data contained in a QCSet. The four main views include the StormTrackView, WindObservationView, GeographyView and the LandmarkView (Figure 10). The StormTrackView (figure 10b) displays all the information about each of the *hrd.meteorology.StormTrackFixes* contained in the current StormTrack. It also provides the user with tools to edit, add (via interpolation, extrapolation, load from database or manual entry) and remove fixes as well as tools for loading entire storm tracks from the database into the current QCSet.

a. 

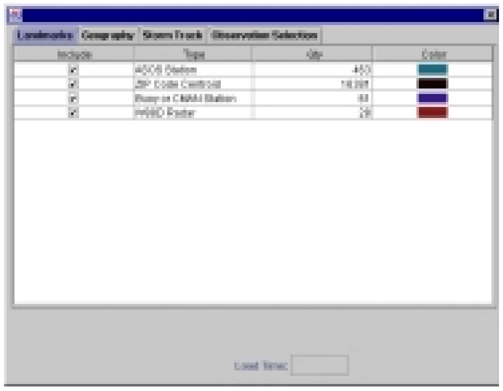
b. 

Lat	Lon	Date	UTC	Speed (mps)	Direction (deg)	Source
25.49	-76.90	8-23-1992	2232			
25.38	-77.63	8-24-1992	0013	8.50 mps	254 deg	
25.43	-78.15	8-24-1992	0212	8.90 mps	283 deg	
25.38	-78.82	8-24-1992	0418	8.83 mps	285 deg	
25.37	-79.32	8-24-1992	0546	8.73 mps	288 deg	
25.43	-79.82	8-24-1992	0700	8.93 mps	282 deg	
25.43	-80.01	8-24-1992	0803	10.34 mps	279 deg	
25.44	-80.16	8-24-1992	0831	8.83 mps	278 deg	
25.45	-80.22	8-24-1992	0845	8.78 mps	278 deg	
25.46	-80.31	8-24-1992	0900	8.87 mps	277 deg	
25.47	-80.42	8-24-1992	0921	8.89 mps	278 deg	
25.48	-80.49	8-24-1992	0936	8.43 mps	277 deg	
25.48	-80.54	8-24-1992	0945	8.77 mps	277 deg	
25.49	-80.62	8-24-1992	1000	8.58 mps	278 deg	
25.50	-80.67	8-24-1992	1018	8.78 mps	277 deg	
25.51	-80.77	8-24-1992	1030	8.88 mps	277 deg	

04:10Z 09:00Z 10:30Z

c. 

Include	Platform	Qty	Time Scope	Time Range	Color
✓	Ship	31	3	08/24 05:40 to 08/24 10:17	■
✓	AirForce Recon	411	3	08/24 04:43 to 08/24 11:33	■
✓	DARDC	6	3	08/24 06:00 to 08/24 11:49	■
✓	CMAN	323	3	08/24 03:00 to 08/31 11:00	■
✓	Buoy	40	3	08/24 05:36 to 08/24 11:36	■

d. 

Include	Type	Qty	Color
✓	ASOS Station	453	■
✓	ZIP Code Centroid	18,201	■
✓	Buoy or CMAN Station	60	■
✓	WSR88D Radar	26	■

Figure 10. The Four DataViews: a) GeographyView, b) StormTrackView, c) WindObservationView and d) LandmarkView.

The WindObservationView (figure 10c) dynamically displays all the different observation platforms represented by the data in the current QCSet (Air Force and NOAA reconnaissance aircraft, satellites, CMAN stations, buoys, ships, airport reports, GPS dropwindsondes, etc.) and details about those data groups. These details include the number of observations currently loaded for a given platform, the time range for the observations, the specified time scope for inclusion into the QCSet, whether or not the data are to be included in the QCSet and the color used to draw data that came from the given platform. The GeographyView (figure 10a) allows the user to select the current basin he wishes to view. This view is only available while in the synoptic plot mode. The Landmark view displays the number and type of landmarks, user defined points of interest to be plotted over the geography, currently loaded into memory as well as their display colors.

4.3 *Use Cases and Sequence Diagrams*

A use case analysis helps to enumerate the scenarios that are fundamental to a system's operation. These scenarios collectively describe the system functions of an application (Booch, 1994). Below I present the main use case diagram for the application and a catalog of its external actors. In use case modeling, "the system is looked upon as a 'black box' that provides use cases" (Erikkson,1998). A use case diagram is primarily used to describe the functional requirements of the system. How this is done is not important at the time that the use case diagram is created. I also go on to describe the use cases which were essential to fulfilling the requirements described earlier and presented in the main use case diagram. Because use cases represent a complete functionality as perceived by an actor, for or on the behalf of an actor, each use case presented pertains to QCSet operations and user interface operations and how they are initiated by or provide value to either the user or the database, the two external actors for the system. These use cases helped formulate Sequence Diagrams (Figures 12-20) later in the analysis. A sequence diagram "reveals the interaction for a specific scenario—a specific interaction between the objects that happens at some point in time during the system's execution (e.g. when a specific function is used)" (Eriksson, 1998). They are essentially a different way of looking at an object diagram with the advantage being that it is easier to read the passing of messages in relative order (Booch, 1994). Because of their affinity, I will present each use case and its corresponding sequence diagram together along with the methods and other sequenced functionality that the sequence diagram helped me to discover.

4.3.1 *Use Case Diagram and Actor Catalog*

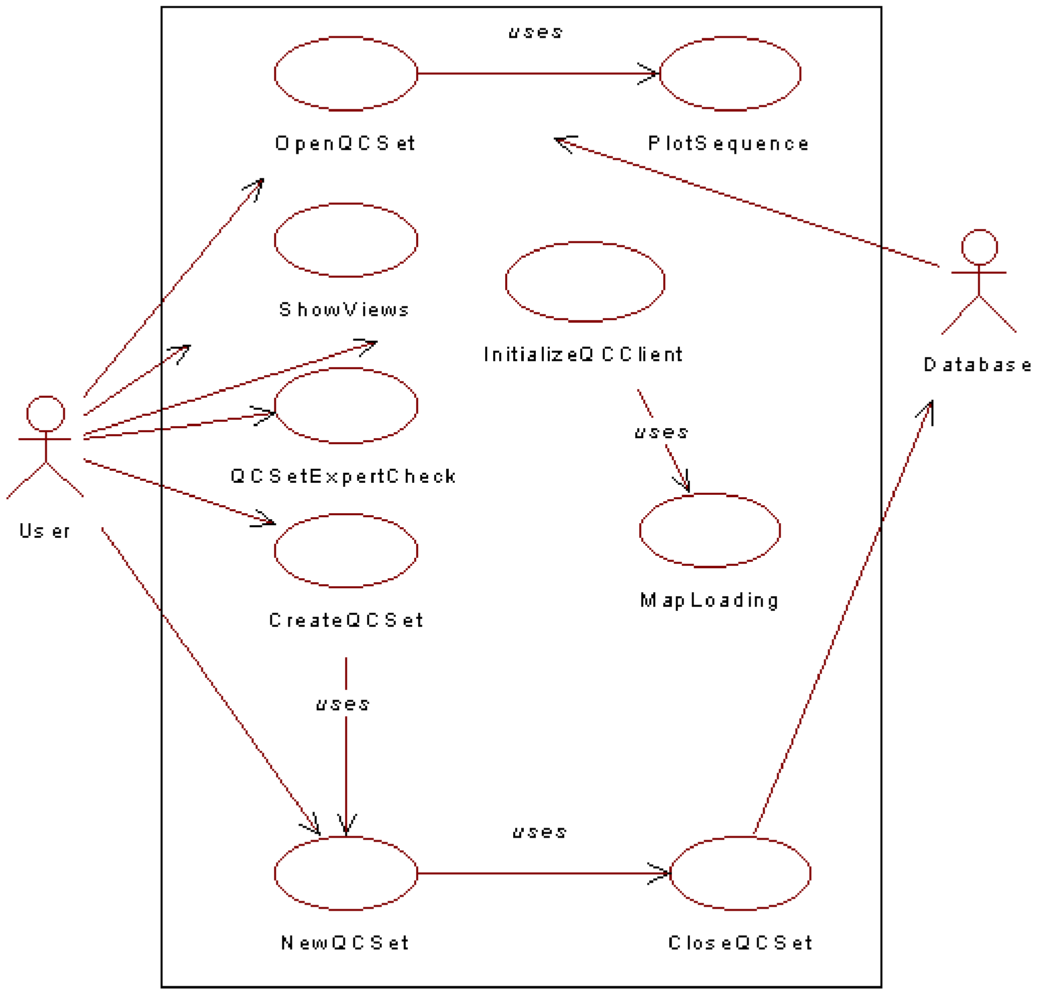


Figure 11. Main Use Case Diagram for H*WIND QCClient application

The two main actors for this system are the User and the Database.

- User: Any person using the H*WIND QCClient interface. The user may be either an operational user or a research user. The system is designed for use by any user on the Internet, but typical users will be research meteorologists from the NOAA's Hurricane Research Division.
- Database: A data store that contains incoming raw and processed surface wind observations, storm track fixes, user data sets, storm tracks and analysis results. The database may be relational, object, object-relational or even a hierarchical system of flat files. Non flat file databases are preferred, however, because of the superior database management features that are typically available for them.

4.3.2 *QCClient Initialization Sequence*

Upon starting the QCClient application, a separate thread is dispatched to load the user’s default basin map at the same time as the rest of the initialization occurs. The QCClient’s initial menu configuration is setup and shown along with an empty canvas. The canvas remains empty while the map continues to load. All the while, however, the user may use those features of the QCClient that do not require the map to be loaded. When the map finishes loading, it is drawn on the canvas. Figure 12 shows this use case’s corresponding Sequence Diagram. The QCClient initialization sequence helped in my design of the *main* method for the QCClient application and also aided in the constructor and initial state implementation for the QCClientFrame class.

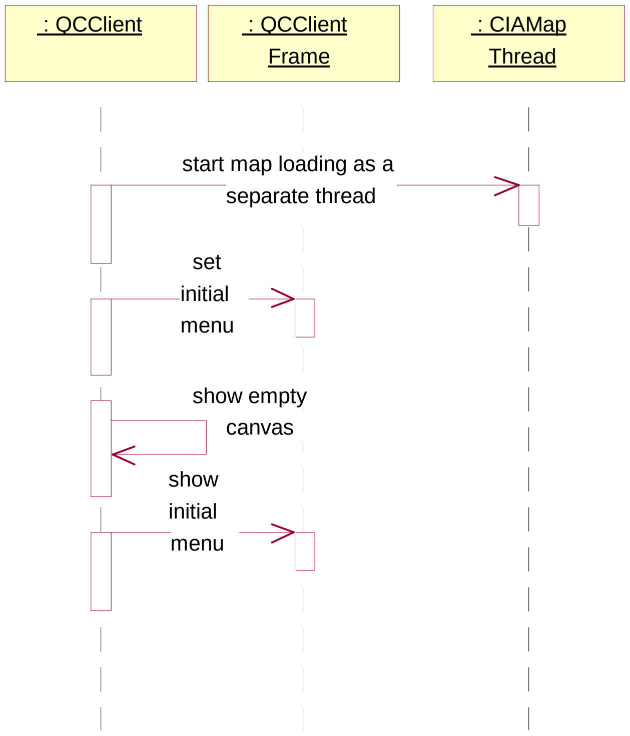


Figure 12. QCClient Initialization Sequence

4.3.3 *CIAMap Loading Sequence*

Loading the map in a separate thread and synchronizing the main thread with the map thread requires careful orchestration. When the call comes for the map to be loaded, the QCClient must instantiate a CIAMapThread object, give it a location from where to load the map (filesystem, database or other URL), and set its callback object and callback method. The CIAMapThread object is then started and proceeds to load the map. When the map finishes loading, the CIAMapThread object runs the callback method in the callback object as specified earlier. This method essentially tells the QCClient that the CIAMapThread has completed its task. It is necessary because there is no other way to know when the map has finished loading other than running a loop in the QCClient that periodically checks the map's status. Such a loop would defeat the purpose of using a separate thread. After the callback notification occurs, the QCClient can set its current view area according to the boundaries of the loaded map. A Sequence Diagram for this use case is presented in Figure 13. The main class' *makeAndLoadMap* and *mapCallback* methods and the entire *CIAMapThread* class were each designed as a result of this sequence diagram.

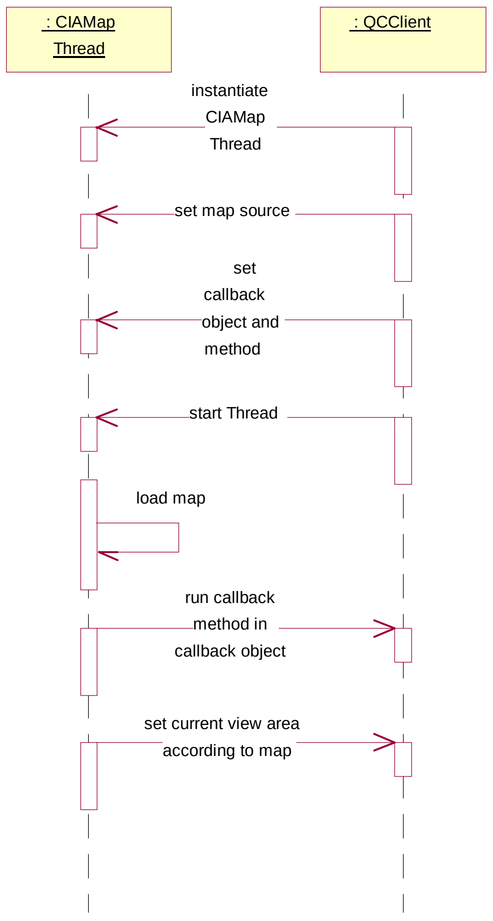

: CIAMap Thread
: QCClient
instantiate CIAMap Thread
set map source
set callback object and method
start Thread
load map
run callback method in callback object
set current view area according to map

Figure 13. Map Loading Sequence

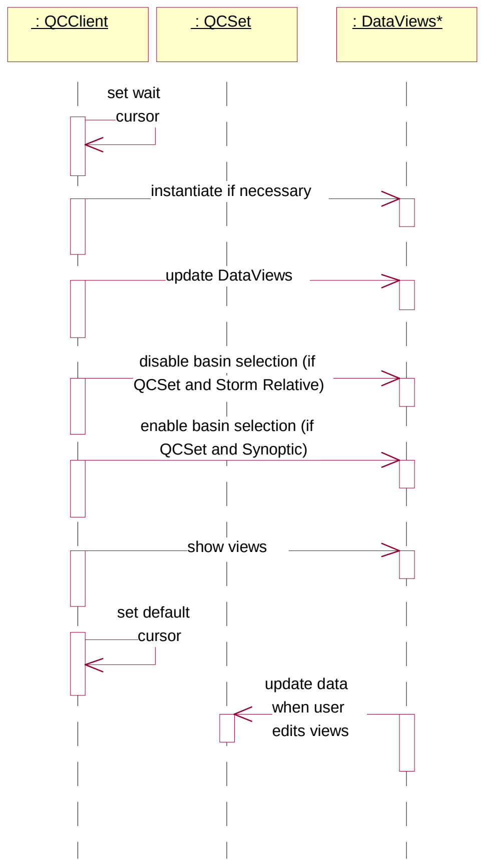
A sample of the code segment required to load a basin map through a thread follows:

```
 // get the location of the map for the currently selected basin
// from a local hash table
String dataLocation =
  ((BasinInfo)(basinTable.get(
    new Integer(selectedBasin)))).getMapLocation();

// tell the map instance where to get the map data
theCIAMap.setSource(dataLocation);
// “mapCallback” is the method that will be called
// from “this” object when the map has finished loading.
theCIAMap.setCallback(this, "mapCallback");
// tell the map instance that it will get the map data
// specified by “dataLocation” from a Java archive
// (JAR file) in the classpath
theCIAMap.setLoadingMethod(CIAMapThread.FILE_RESOURCE);

// setup and start the Thread… this will begin
// loading the map
Thread mapThread = new Thread(theCIAMap);
mapThread.start();
```

4.3.4 *Show Views Sequence*

Before presenting the use case for showing the main data views, I will first discuss the general approach I took to separating data from their respective views. Using Java’s listener interfaces helped make data abstraction more automated. When implementing such a listener, the designer can designate a delegate object for handling certain types of events (mouse events, window events, state changes, data change events, etc.) in addition to assigning a listener to the object in which those events will occur. For example, when a user edits a graphical table representing the view of a storm track, a listener triggers the abstract model for that data to change the underlying data accordingly. The methods for actually changing the data, a detailed and application specific task, must be written by the developer. Using these techniques, the designer can automate a Model-View-Controller (MVC) architecture to separate data from views of those data. I chose to use the listeners but not to the extent that they controlled everything about the application. I found that occasionally providing my own update methods and making my own models instead of using Java’s default abstract models, strangely enough, kept the design simpler and easier. This is, of course, purely a matter of taste, and others may find that a complete immersion into Java’s listener-model framework works better for them.

This use case describes the actions taken when the user wishes to invoke the views window for the current data set (QCSet). First, the DataViews object and all of its aggregate views and objects should be instantiated if they are not already. As a general rule, to help reduce memory usage and application startup time, objects are not instantiated until they are needed. Secondly, all the views must be updated. This is where my design and the Java MVC architecture diverge. If I were to strictly employ Java’s MVC methods, the views would be automatically updated and no separate “updateViews” method would be necessary although the logic behind the update needs to be written in either case. My version essentially defers the updates. After the views are updated, the basin map selection interface in the GeographyView must be enabled if the current plot mode is set to a synoptic (earth relative) coordinate system or disabled if set to a storm relative coordinate system. The views are then presented to the user so that he may examine and edit the data that they represent. See Figure 14 for a Sequence Diagram relating to this use case. This sequence helped me in planning how to accomplish the tasks in the *QCClientFrame’s showViewsAction* method in the midst of all the other sequences occurring concurrently in the system.


: QCClient
: QCSet
: DataViews*
set wait cursor
instantiate if necessary
update DataViews
disable basin selection (if QCSet and Storm Relative)
enable basin selection (if QCSet and Synoptic)
show views
set default cursor
update data when user edits views

*Represents several objects: WindObservationView, StormTrackView, GeographyView, LandmarkView

Figure 14. Show Views Sequence

4.3.5 *Plot Sequence*

This use case describes the order in which the QCClient draws a plot and all of its components. The plot sequence begins by checking for a loaded CIAMap. If a map is not loaded, the map and rest of the plot components are not drawn. In the likely event that a map is loaded, the CIAMapArtist is called to draw the map portion indicated by the user's current zoom area onto the canvas. After the map is drawn, and if a QCSet is loaded, the StormTrackArtist and WindObservationArtist are called to draw the storm track and wind observations respectively. Any Landmarks that are loaded are then drawn regardless of a whether or not a QCSet is present. Figure 15 presents the Sequence Diagram for the typical plot sequence.

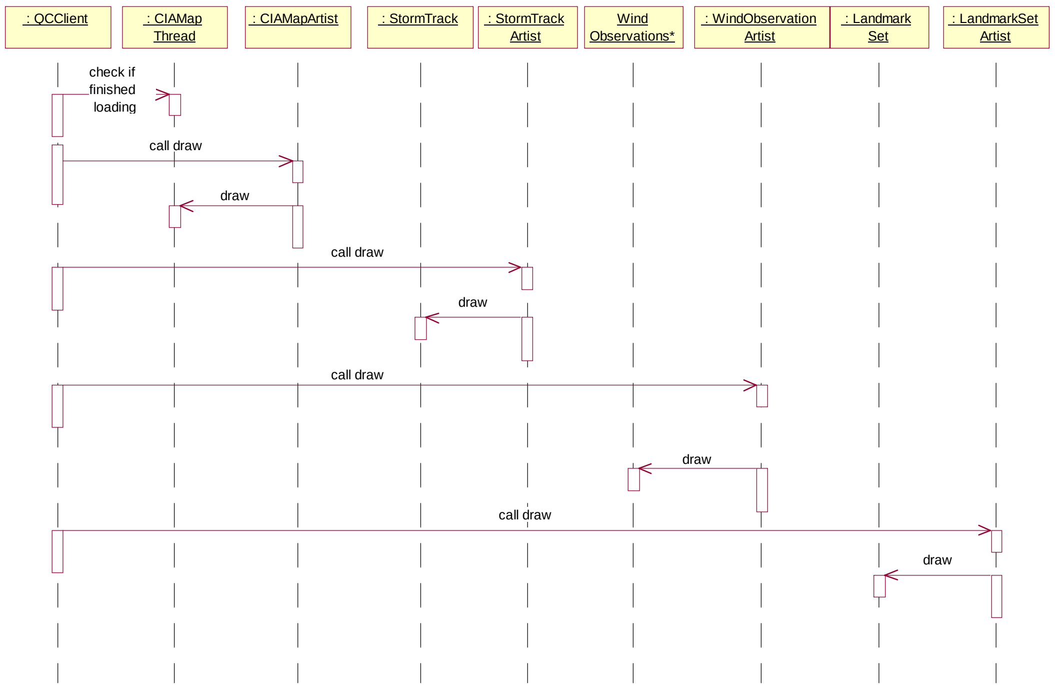


Figure 15. Plot Sequence

A great deal of work goes on inside the individual Artists' draw methods to properly draw their components. Each Artist must transform the map coordinate system into the canvas' coordinate system and draw each component accordingly. Each Artist, in turn, has its own responsibilities specific to drawing its specified component. Most notably, the WindObservationArtist must perform a number of scaling, rotation and translation transformations to draw wind barbs at the correct angle according to the wind direction and draw the appropriate number and shape of flags on the barb shafts to indicate the windspeed for that observation. Although interesting, the specific scenarios for the individual Artists are not represented in any Sequence Diagrams. The basic sequence taking place within the QCClient's *paint*, *repaint* and *refresh* methods hinges on the messages presented in this sequence.

4.3.6 *QCSet Creation Sequence*

The "QCSet Creation" use case describes a scenario in which a user has the QCClient running and wishes to create an entirely new QCSet. The application is assumed to have no QCSet currently loaded, so the user must indicate that he wants a new QCSet by selecting the appropriate menu option. An empty QCSet is created and the menu is updated to reflect that state. The user would then typically open the DataViews and use the views to add and load storm tracks, storm fixes, and wind observations from the database and/or filesystem. He may then edit and manipulate the data through the views, as well. Figure 16 shows the Sequence Diagram for such a scenario. This sequence particularly helped me visualize what needed to be available to the user (via GUI or otherwise) during the creation of a QCSet.

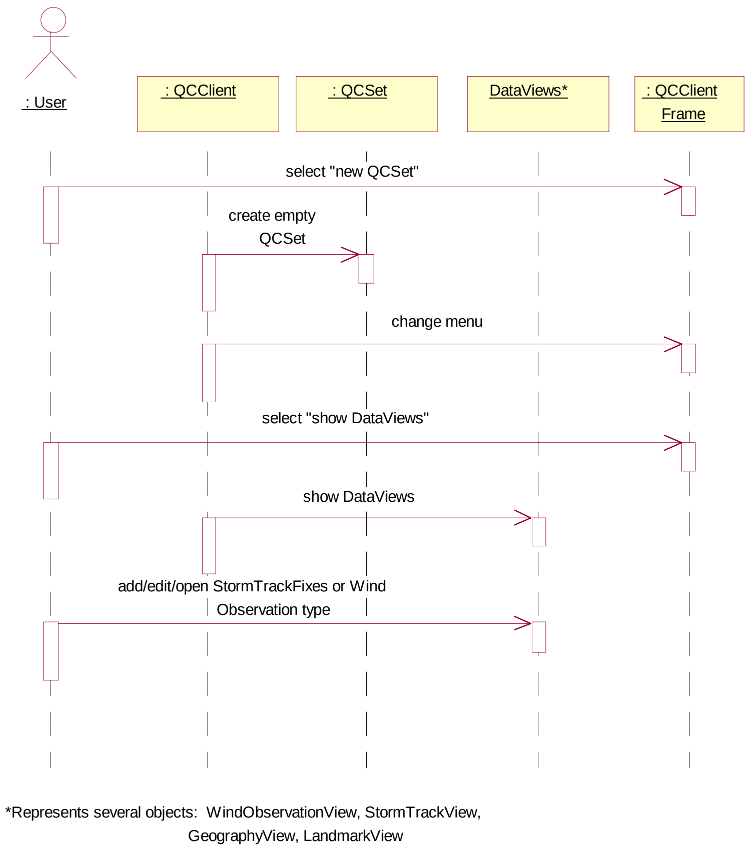

: User
: QCClient
: QCSet
DataViews*
: QCClient Frame
select "new QCSet"
create empty QCSet
change menu
select "show DataViews"
show DataViews
add/edit/open StormTrackFixes or Wind Observation type
*Represents several objects: WindObservationView, StormTrackView, GeographyView, LandmarkView

Figure 16. Create QCSet Sequence

4.3.7 *New QCSet Sequence*

The "New QCSet Sequence" differs from "QCSet Creation Sequence" in that it only pertains to the sequence of events that occur when a user indicates that he wishes to create a new QCSet, not the overall sequence of events that precede and follow that action. In other words, it describes what happens when the user selects the menu option for a new QCSet; nothing else. After the user selects the menu option, the application performs a close operation on the currently loaded QCSet (see "*Typical QCSet Close Sequence*") and the menu is updated to reflect the current state. If a map is not currently loaded, the application "spins" and waits for the map to finish loading or stops after a prescribed time-out period. A new, empty QCSet is then instantiated and loaded and the menu is changed to reflect the state of the application. See Figure 17 for the Sequence Diagram relating to this use case. The *QCClientFrame* class' *selectedNew* method accurately captures the new QCSet sequence.

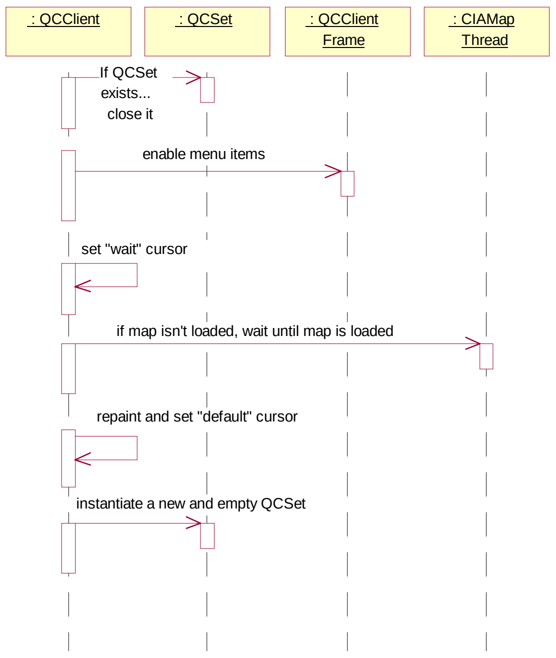

: QCClient
: QCSet
: QCClient Frame
: CIAMap Thread
If QCSet exists... close it
enable menu items
set "wait" cursor
if map isn't loaded, wait until map is loaded
repaint and set "default" cursor
instantiate a new and empty QCSet

Figure 17. New QCSet Sequence

4.3.8 *QCSet Open Sequence*

The sequence of events for loading a previously archived QCSet begins with a check for an open QCSet. If one is loaded, it is closed with the user's permission and the user is then presented with the specialized GUI for selecting a QCSet to load. The selected QCSet is then loaded from the database and its components are plotted on the canvas. The menu is then set to reflect the state of the application. Figure 18 shows the

Sequence Diagram for this use case. The open QCSet sequence helped me formulate the *QCClientFrame's selectedOpen* method in addition to helping me to discover how to set the zoom stack and various other display aids upon loading a QCSet.

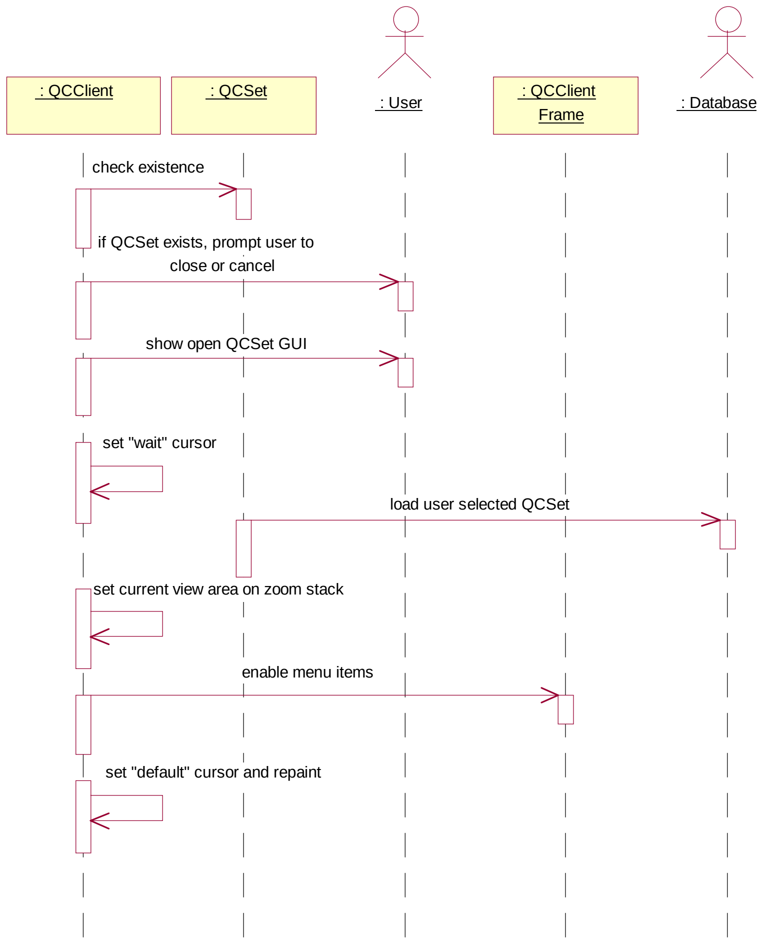


Figure 18. Open QCSet Sequence

4.3.9 *QCSet Close Sequence*

In order to close a QCSet, one must, naturally, be open in memory. If a QCSet indeed exists, a check is made to see if it is in an altered or "uncommitted" state. If the

QCSet is altered, the application prompts the user to commit the QCSet to the database. This is similar to saving a file in a traditional filesystem based application. After the commit stage, the QCSet is freed, the menu changes to reflect that state and the plot is redrawn. The data views are updated and the basin map selection interface in the GeographyView is disabled. See Figure 19 for this use cases' Sequence Diagram. The close QCSet sequence provided an excellent guide for developing the *selectedClose* and *selectedExit* methods in the *QCClientFrame* class. It also helped spell out the role of the commit functionality in the application and provided a clear sequence for how to reset the state of the application after the removal of a QCSet from the system.

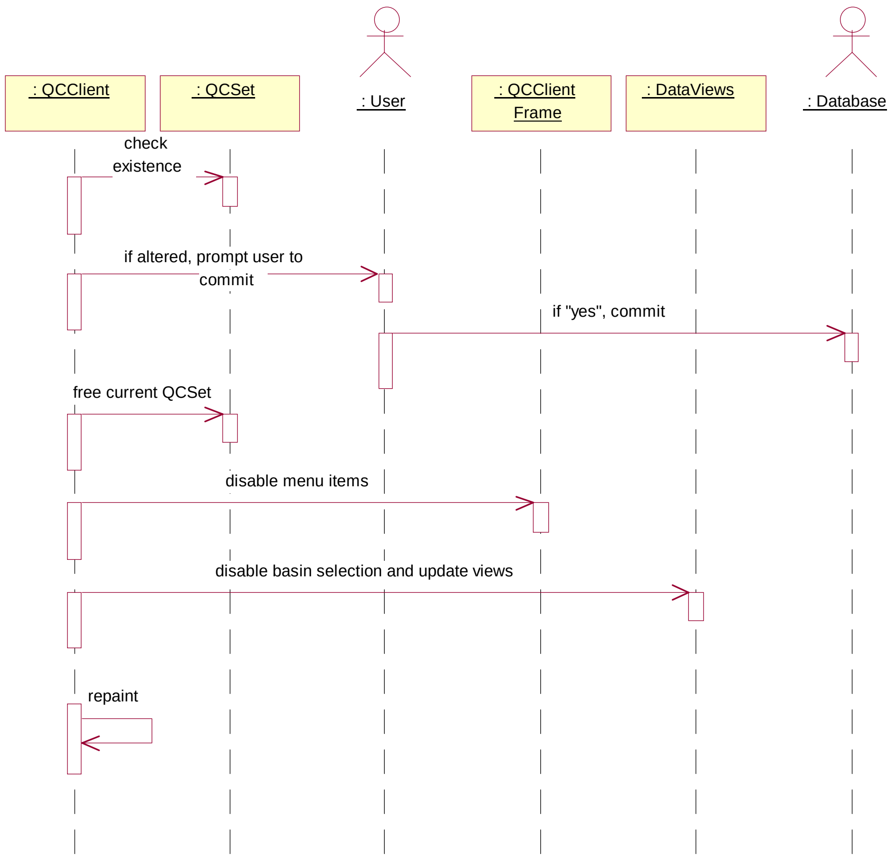


Figure 19. Close QCSet Sequence

4.3.10 *QCSet Expert Check Sequence*

The "Expert Check" use case describes the actions behind an automated expert check of the QCSet. Note that the class responsible for running the expert checks, QCSetValidator, is an abstract class and may not be instantiated, but its methods may be used at the class level. When called to run the checks, the QCSetValidator methods issue and attach "notices", QCSetError objects, to the QCSet being checked. The application then displays a panel showing any notices for the current QCSet and the plot is redrawn differentiating the observations in question from others. The user may then view the details of the errors by selecting the entries in the error panel. Figure 20 presents the Sequence Diagram for the QCSet validation sequence. The expert check sequence provides the sequence necessary to understand the interaction between the *QCSetValidator* and *QCSetError* classes and the QCSet which contains them. In the future, this sequence will aid in the development of the user interface described above.

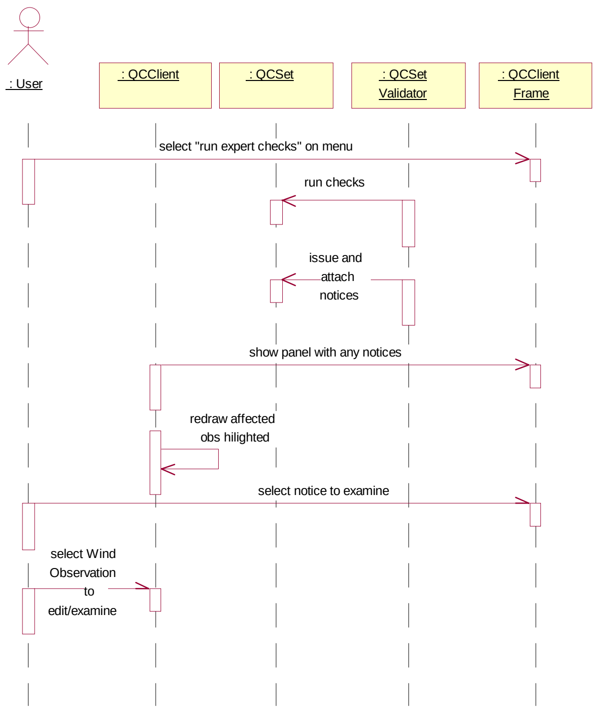

: User
: QCClient
: QCSet
: QCSet Validator
: QCClient Frame
select "run expert checks" on menu
run checks
issue and attach notices
show panel with any notices
redraw affected obs hilighted
select notice to examine
select Wind Observation to edit/examine

Figure 20. Expert Check Sequence

4.4 *State Diagrams*

State Diagrams are useful for capturing the view of the dynamic model of a single class. These diagrams are state transition diagrams which show how a class moves from one state to another and the results of such a change. Only those classes that exhibit some significant event ordered behavior require state diagrams. State Diagrams are also helpful for discovering a class' additional services. Below I present State Diagrams for the QCClient, QCSet, QCSetValidator and DataViews classes. Since the QCClient is the main class in the application, I used its State Diagram (Figure 21) during the design stages of the project as if it was a system wide state transition diagram. This main diagram served as an invaluable tool that helped me gain an understanding of the dynamics of the system I was about to build. The QCSet diagram (Figure 22) aided in visualizing how the QCSet would be affected by the application at different times. This, in turn, helped me discover which methods needed to be invoked in each particular state as well as which additional methods needed to be designed for proper execution in those states. For example, when the user closes a QCSet, it could be in one of several states and the problem is compounded when one considers exiting the application. Other states such as *valid* and *invalid* can and will be added to aid in the design of the automated quality control system. The QCSetValidator State Diagram (Figure 23), along with a revamped QCSet State Diagram, will serve as a guideline for the development of a complex automated quality control subsystem. The QCSetValidator diagram enumerated the various states that the validator will be in during a systematic check of an arbitrary QCSet. Ordered states and actions are interchangeable in this and potentially many State Diagrams because of the deterministic way in which the QCSetValidator, in this case, moves from state to state and how its transitions closely follow a typical flow of execution. The DataView’s State Diagram (Figure 24) is similar to that of the QCSet class because the DataViews merely represent the data that are stored in a QCSet.

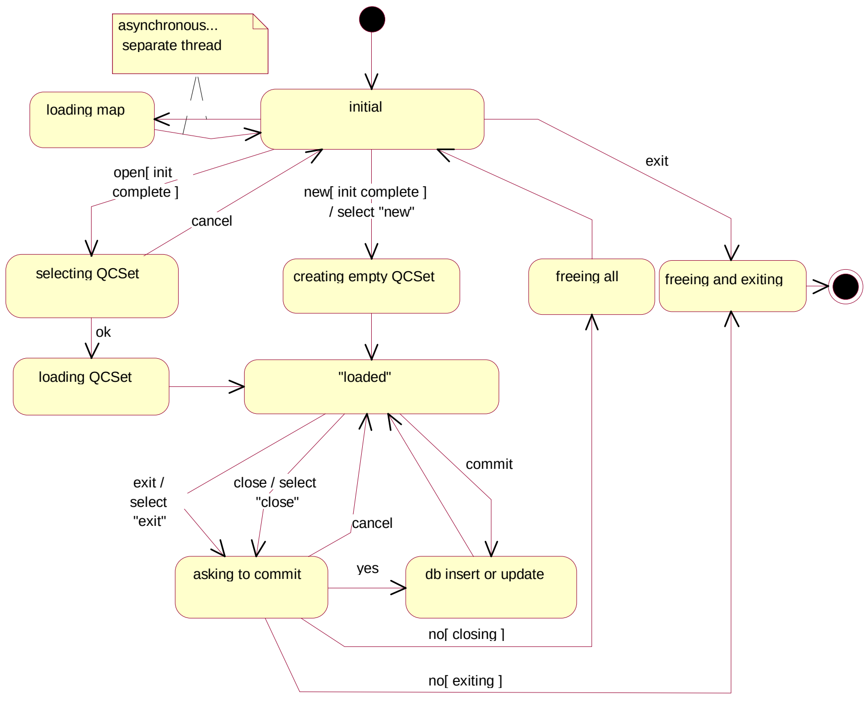


Figure 21. QCClient State Diagram

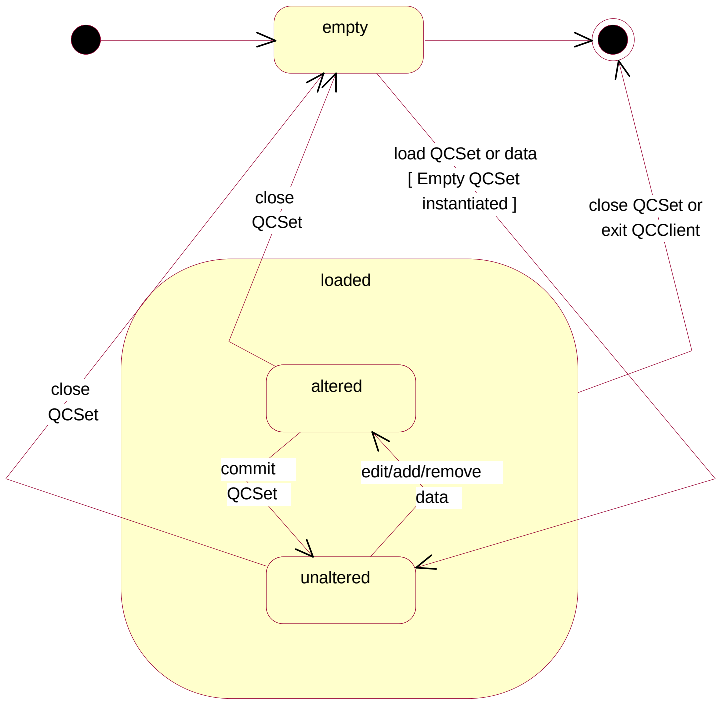


Figure 22. QCSet State Diagram

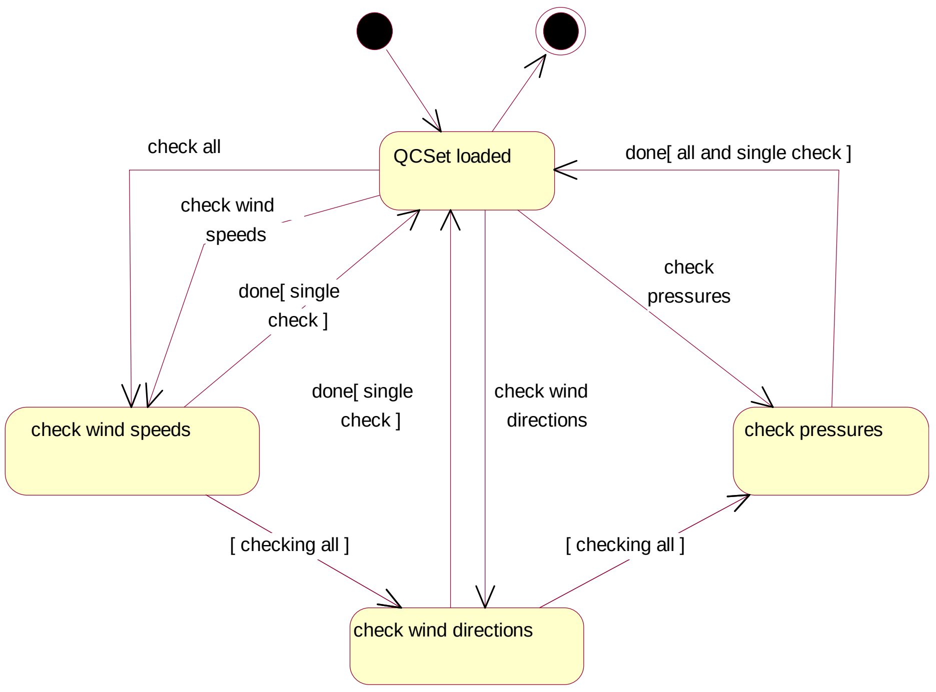


Figure 23. QCSetValidator State Diagram

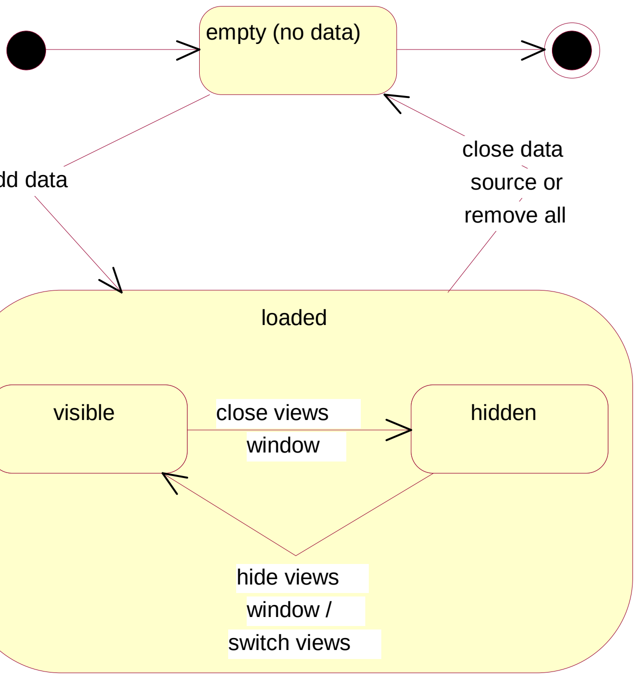

empty (no data)
add data
close data source or remove all
loaded
visible
close views window
hidden
hide views window / switch views

Figure 24. DataViews State Diagram. Similar to that of QCSet (Figure 22).

4.5 *Interface Requirements*

The QCClient's interface required several purely conceptual characteristics independent of the particular implementation environment. Design requirements for the user interface addressing these characteristics include:

- The application should be centered around the plot area. Everything should be designed around it and it should be contained within the application's main screen.
- The data and the user views of the data should be separated as much as possible as in the Smalltalk MVC (Model-View-Controller) architecture.
- Any change in the data or the user preferences should be immediately reflected in the plot. This implies a separate window for the data views.
- Only one QCSet should be loaded at a time. In other words, a single document interface should be used. This is particularly important when one considers that the plot and the data views will be contained on separate screens. Programmer and user confusion would be certain to arise given a multiple document/multiple screen scenario.
- All database and filesystem queries should be made through the use of a helper screen or "wizard" interface. This screen should dynamically guide the user through the specifics of the query and eventually extract the needed information to make the query. No free form queries will be allowed. This is important for several reasons: overall database security, QCSet archive security, ease of use and added abstraction between the application and database tiers.
- In general, the user should have to type as little as possible in an effort to minimize errors and error checking code.

5. **Implementation Details**

Although much of these details are, traditionally speaking, design issues, they focus heavily on the implementation side of design and fall well beyond the realm of analysis. When considering a development process with a blurred and indistinct separation of analysis and design, implementation details such as these should probably be given their own stage. I will use this section to describe specifically how and why I did certain things given the constraints of the chosen implementation OOPL, Java, and other constraints such as those discussed in the *Analysis and Design* section above.

5.1 *Java*

I chose Java as the primary implementation language for H*WIND not only because of its portability, but also, and possibly more importantly, because of its highly integrated and abstract solutions to everyday programming problems. The Java development environment provides designers and programmers with tools for developing applets or applications that deal with many low level and specialized concepts at an abstract level common to most modern operating systems. The core Java API provides tools for programming with threads, networking, GUI implementation, database connectivity, file i/o and object distribution, to name a few. Two of these features, database connectivity and a specific extension of the GUI API (the Java Foundation Classes' *Swing* components), are discussed at greater length below in separate sections.

In addition to features in the core API, Java also improves over C++ as an OO programming language in several ways. Java, unlike C++, was designed to be an Object Oriented language from the ground up (Flanagan, 4). Most things in Java are objects; the only exceptions are primitive types such as the numeric, byte and boolean types. Although it is designed to look like C++, many of the complexities of that language are not present in Java. For example, all method parameters in Java are passed by implicit reference rather than by value or by pointer. Java also does its own automatic garbage collection. In order to delete an object, one only needs to assign a null value to a reference to that object. The garbage collector will then free the object if no other references to that object exist. Java also provides built in exception handling which, in combination with the previously named features and others, yields more robust programs.

5.1.1 *Database Connectivity*

All database connectivity in H*WIND is accomplished through the use of the Java Database Connectivity (JDBC) API. The JDBC specification is a basis for developers to interface with a variety of data sources. The main idea behind it is that a standard high level interface is used to access any database (in our case, an Oracle 8 database) and the low level JDBC driver manages the connections and queries. JDBC drivers for specific databases are typically provided by the manufacturers of those databases.

In addition to an abstract database interface, HRD has limited access to the H*WIND schema through stored database procedures. The application employs two packages of database side PL/SQL procedures; one for queries and one for actions such as insertions and updates against the schema. These procedures serve to protect the database's integrity against both malicious and inadvertent sabotage by H*WIND's users as well as by application developers. Each of the query procedures uses one or more cursors for data delivery. Static (class level) JDBC methods call the database procedures and return Java objects to the rest of the application as needed as Figure 25 illustrates.

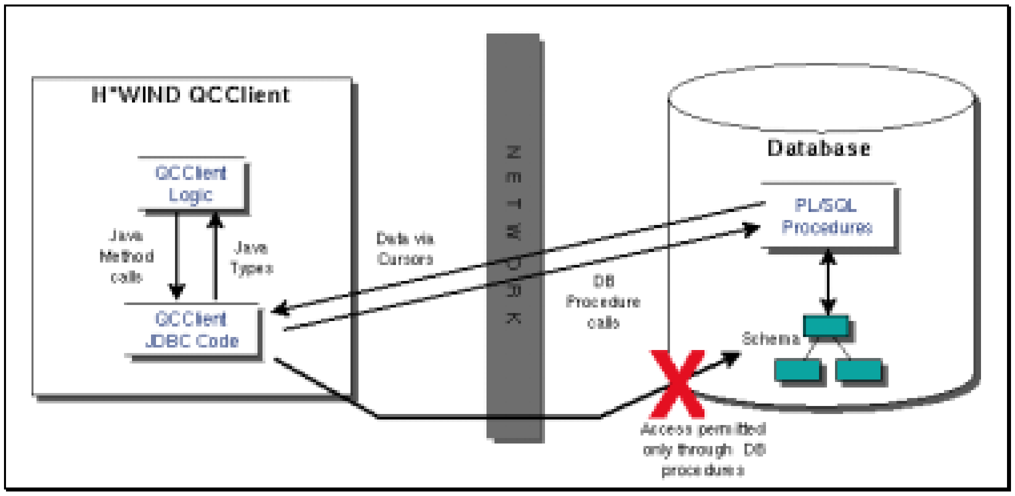
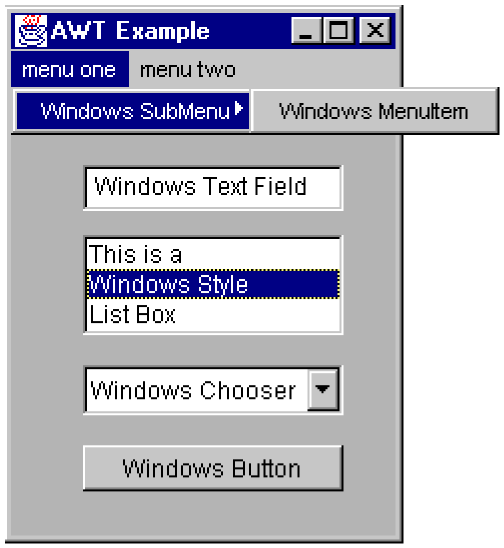


Figure 25. The generalized mechanics of a database query in H*WIND.

5.1.2 *Abstract Windowing Toolkit (AWT) and the Java Foundation Classes (JFC)*

In theory, a Java application designer needs to design a user interface for an application only once because Java's abstract cross platform APIs extend to the Graphical User Interface (GUI) of an application. This abstract GUI code is called the Abstract Windowing Toolkit (AWT). When user code calls for a basic frame window with a menu bar, for example, the Java interpreter for the specific platform under which the application is currently running, makes calls to the local interface API resulting in a frame with a menu bar in the style of the local user environment. Figure 26 illustrates such a call and its results under two different user environments. The UI components that come from the basic AWT are called *heavyweight* components because they involve the use of operating system specific peer components to deliver the desired UI effect.

a. 

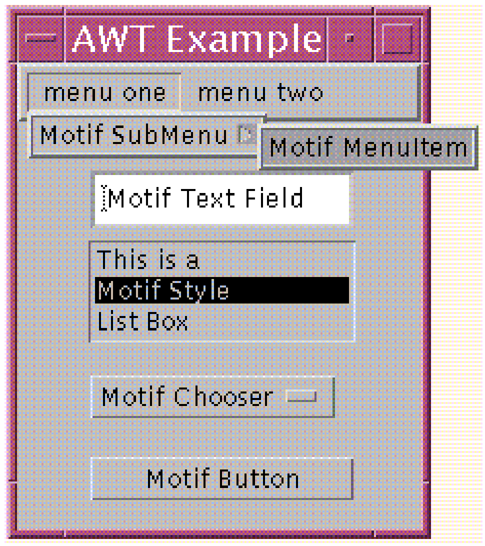
b. 

Figure 26. An example of AWT heavyweight components under a Windows environment (a) and a Motif environment (b). Both figures were created with the same code, but components are rendered differently according to the underlying windowing framework.

While the AWT and its system of calls to peer components is an excellent idea sufficient for most simple applications or applets, it does have some limitations. Applications which use the basic AWT components often need to be massaged or tweaked in order to get a somewhat consistent look and feel across different platforms. In fact, an acceptably consistent look and feel is not easily, if at all, attainable when not taken into account during the early design stages of an application. In addition to basic look and feel problems, because all of the components used in the AWT are heavyweight, they must be available in all the environments for which a Java interpreter will be built.

In other words, many convenient components that users have become accustomed to, such as tabbed panes, tree views, and tables, are not included in the basic AWT for lack of a peer component on some operating systems.

The GUI portion of the Java Foundation Classes (JFC), code named “Swing”, addresses inconsistencies in user interfaces by providing *lightweight* alternatives to the AWT’s heavyweight components. These lightweight components do not require OS specific peers and look exactly the same when rendered under any user environment. The Swing components also provide many of the convenient components that were omitted in the original AWT. It even allows for “pluggable looks and feels” in order to enable users or developers to switch to particular looks and feels whenever they want. Motif, Windows, Macintosh and several original looks and feels are all available with current Swing releases. The H*WIND QCClient GUI is currently a mixture of many lightweight Swing components and a small number of heavyweight AWT components. Eventually, I would hope that all the components used in the QCClient would be lightweight, and, therefore, visibly indistinguishable when run under different user interface environments.

5.2 *Notable Implementation Descriptions*

5.2.1 *Plotting*

All plotting in the QCClient is accomplished through the use of various *hrd.awt.Artist* concrete subclasses. Each Artist is a specialized class responsible for rendering a particular plot component in the QCClient's plot window. The abstract superclass Artist naturally contains attributes that are common to each of its subclasses. It must contain the default and current color, font and scale factor to be used for plotting a component as well as a draw method which is overridden by each subclass. If it were not for its attributes, the Artist class could very well be an interface defining a common draw method. Although I will discuss Artists in the context of rendering their subjects in the H*WIND, they were designed to be able to draw on any Java AWT component. The QCClient needs four Artists to present a plot. These are the CIAMapArtist, the StormTrackArtist, the WindObservationArtist and the LandmarkArtist. All except the LandmarkArtist are discussed in depth in the following paragraphs.

The CIAMapArtist is called first to render the user's currently selected geography. The coastlines represented by the CIAMapPosition objects contained by the CIAMap must be plotted by drawing a line from each position to the next position. Every position contains a "groupNumber" attribute which serves to differentiate land masses. For example, all CIAMapPosition objects for any given island will have the same groupNumber. Because the plot represents a particular *hrd.geography.GlobalArea* stored at the top of the zoom stack, each Artist must calculate two dimensional horizontal and vertical scaling factors based on the current canvas size and the current minimum and maximum visible latitude and longitude coordinates. After plotting the contiguous map positions, the CIAMapArtist moves on to drawing latitude and longitude grid lines and labels them appropriately.

If a QCSet is loaded and it contains a storm track, the StormTrackArtist is called to render that track. This Artist is similar to the CIAMapArtist in that it draws a series of lines between adjacent points in a list. Two of the track's fixes respectively indicate the user's choices of begin and end times for the inclusion of data in the plot. Figure 27 illustrates this functionality. Concentric blue circles are drawn around the user's choice for the center fix, the fix around which a potential analysis will be performed. In the storm relative plot mode, all the wind barbs are plotted relative to the storm at the time and position indicated by the storm track fix selected as the current center. Code is also in place to allow the drawing of labels next to the fixes in the track.

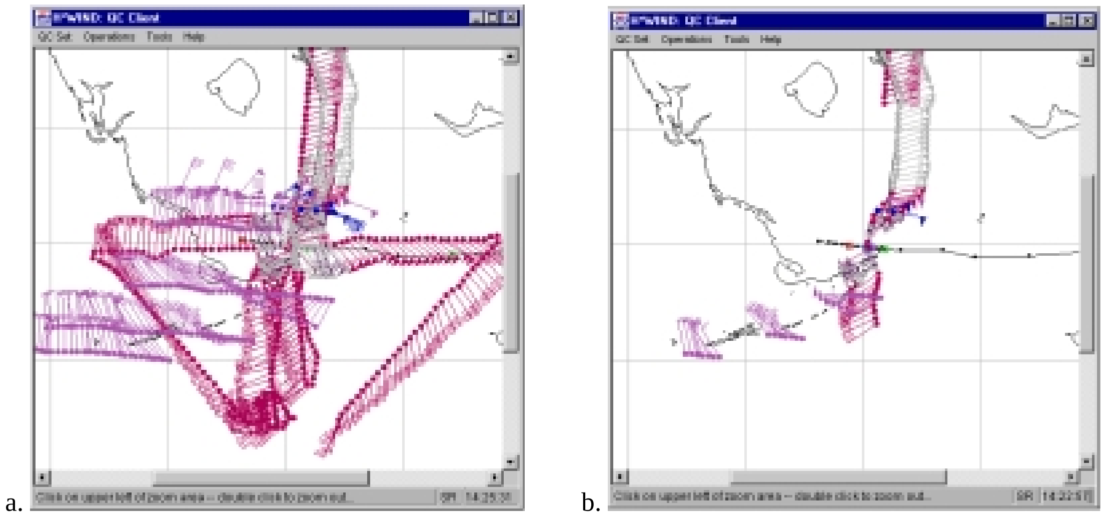


Figure 27. Examples of data filtering using different begin and end times in the QCSet's StormTrack. Both plots are centered around the 0900 UTC fix for August 24th 1992. Figure 2a: begin time = 0410 UTC; end time = 1030 UTC. Figure 2b: begin time = 0831 UTC; end time = 0935 UTC.

The WindObservationArtist is the next to be called in the plotting sequence. It is more intricate than the other Artists because it must perform such a great deal of complex work in a short period of time. In general, the WindObservationArtist must loop through a list of lists where each sublist contains observations from common platform types (Air Force reconnaissance, Buoys, etc.). For each observation it comes across in the loop, the Artist must first determine the *hrd.geography.GlobalPosition* at which to plot the observation. If the current plot mode is set to “synoptic”, then the observation is plotted at its original position. However, if the plot mode is set to “storm relative”, the observation is plotted at its position relative to the current center fix (see Figure 4). If the storm relative positions for the current center have not been calculated, the WindObservationArtist asks that they be calculated before it plots any storm relative observations. Rendering an observation in the plot window requires drawing a wind barb (Figure 5) representing the speed and direction of the current observation in the color and style specified for the current platform and observation status (passed, failed or edited). The Artist performs a series of matrix calculations to quickly and efficiently rotate and translate each barb segment into place and to rotate and translate the barb itself into its final position in the plot window. Although Java now provides a new specialized 2D API for these types of manipulations, I was forced to use my own matrix and vector classes (*hrd.math.Matrix* and *hrd.math.MathVector*) because of the slow drawing performance I experienced when using the 2D API.

5.2.2 *Wizards*

As stated previously in the section entitled *User Interface Requirements*, an H*WIND user should have to type as little as possible in an effort to minimize errors and programmatic error checking. To help realize this requirement, I developed the abstract *hrd.awt.Wizard* class. Subclasses of the Wizard class help to guide users through conceptually simple yet programmatically difficult tasks such as formulating queries involving the database or filesystem, adding StormTrackFixes and setting parameter values for an automated analysis. In general, a wizard is designed to return a result. This result is of type Object, and, thus, must be cast to whatever the specific subclass intends to return. For example, the QCSetQueryWizard returns an object of type *java.lang.Integer* to represent the unique database id for a desired QCSet while the AnalysisWizard returns a compound object containing all of the user's parameters for a particular analysis run.

Wizards implement the familiar "next", "back", "finished" and "cancel" buttons to traverse the potentially many panels required to arrive at a result. Each of these panels can be thought of as a state in a State Diagram and the user's selections or inputs in each panel determine the appropriate transition to the next state. Figures 28 and 29 show the similarities between what the user sees when using a wizard and that wizard's State Diagram.

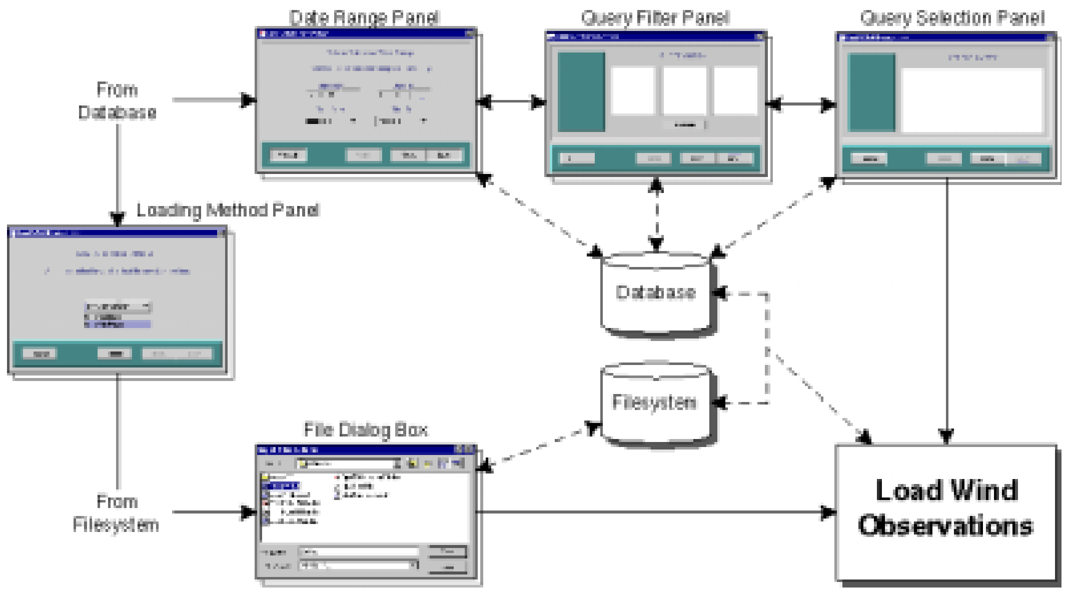


Figure 28. Logical flow of the WindObservationQueryWizard is similar to its State Diagram.

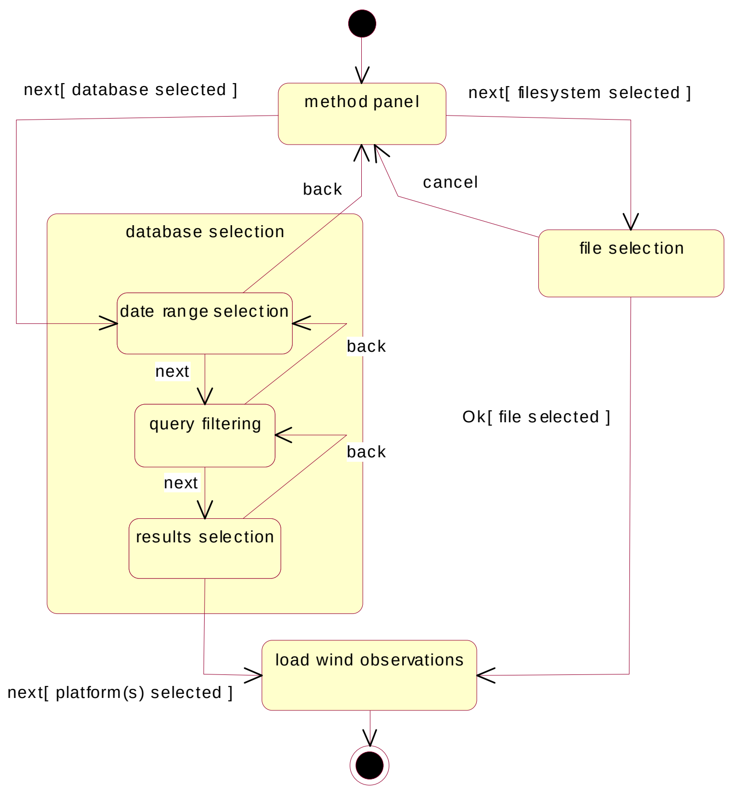


Figure 29. State Diagram for the WindObservationQueryWizard is similar to its flow diagram.

6. **Conclusion**

In developing H*WIND and its QCClient interface, I have essentially built a specialized Geographic Information System for the real-time storage, retrieval, graphical quality control, inspection and analysis of hurricane surface wind observations. Unlike its filesystem based predecessor, WANDA ("Wind Analysis Distributed Application"), the front end and database interface for this system are portable across various architectures and accessible over Intranet/Internet connections. Furthermore, the use of a database management system facilitates the use of this application for research purposes. The project employed an Object Oriented iterative development method from beginning to end and features the Java programming language to addresses system requirements in an abstract and, thereby, highly portable fashion. The project also prompted the development of the "hrd packages"; various groupings of code that each serve similar interests or deal with addressing the same type of problem. Although these classes are highly integrated, they are not strongly coupled and may be used or extended for use in various other applications.

H*WIND represents an important step in the development of an automated method for the real-time assimilation and objective analysis of surface wind fields. The steady improvement of the timeliness of wind field analyses will lead to quicker responses to disasters and more effective and accurate advisories. Through the use of H*WIND, researchers will be able to develop methods for extracting better and more accurate data samples for smaller storm track based time windows. More data for a smaller time window would provide a more accurate snapshot of a storm's wind field for a given center time and position than what is currently available.

Although H*WIND provides researchers and operational users with an excellent tool for working with wind observations, I have made provisions for several foreseeable improvements to the system. Possible improvements directly affecting the user interface might include scaleable plot features, additional fields for inspection, additional data and data views, a multiple document (data set) interface, the incorporation of persistent user preferences and various built in macros for arriving at derived quantities such as automatic wind direction or windspeed calculation based on other data sets for data that lack a critical windspeed vector component (i.e. radar data or SSM/I satellite data). Improvements should also be made to the semi-automated expert check system. Currently, the system only runs tolerance checks on several scalar values of individual wind observations such as temperature, pressure and wind speed. The expert check system should be extended to make comparisons on all available data as part of a storm system rather than as individual unrelated observations. Perhaps the most seemingly formidable improvement task would be in upgrading entire H*WIND system for the quality control and analysis of tropical cyclones in three spatial dimensions. The current software framework, however, provides an excellent base for such an endeavor. The classes in place today representing two dimensional objects can be extended through inheritance to represent three dimensional objects and the legacy analysis code presently supports analyses of vertical and horizontal cross sections of a data set at different significant levels. All of these improvements are feasible when given today's computing power, but some, multiple data sets and three dimensional visualization in particular, would require significant memory space on client machines.

7. **References**

Amat, Jr., L. R., Powell, M. D., Houston, S. H., Morisseau-Leroy, N., WANDA, HRD's Realtime Tropical Cyclone Wind Analysis Distributed Application. 14th Conference on Interactive Information and Processing Systems, American Meteorological Society, Phoenix, AZ, Jan. 11-16, 1998.

Booch, G., *Object-Oriented Analysis and Design with Applications, Second Edition*, Benjamin Cummings, New York, 1994.

Brooks, Jr., F. P., *The Mythical Man-Month: Essays on Software Engineering – Anniversary Edition*, Addison-Wesley, New York, 1995.

Burpee, R. W., Aberson, S. D., Black, P. G., DeMaria, M., Franklin, J. L., Griffin, J. S., Houston, S. H., Kaplan, J., Lord, S. J., Marks, Jr., F. D., Powell, M. D., and Willoughby, H. E., 1994: Real-time guidance provided by NOAA's Hurricane Research Division to forecasters during Emily of 1993. *Bulletin of the American Meteorological Society*., 75, 1765-1783.

Eriksson, H., Penker, M., *UML Toolkit*, John Wiley & Sons, New York, 1998.

Flanagan, D., *Java in a Nutshell, A Desktop Quick Reference, Second Edition*, O'Reilly & Associates, Sebastopol, 1997.

Jacobsen, I., Christerson, M., Jonsson, P., Övergaard, G., *Object Oriented Software Engineering: A Use Case Driven Approach*, Addison-Wesley, New York, 1996.

Powell, M. D., Houston, S. H., Amat, L. R., and Morisseau-Leroy, N., 1997: The HRD hurricane wind analysis system. 8th U.S. National Conference on Wind Engineering, Baltimore, MD., June 5-7, 1997. Also to appear in *J. Wind Engineering and Indust. Aerodynamics* (in press).

Morisseau-Leroy, N., Atmospheric Observations, Analyses, and The World Wide Web Using a Semantic Database, Master Thesis, School of Computer Sciences, Florida International University, Miami, FL, 1997.

Orfali, R., Harkey, D., Edwards, J., The Essential Distributed Objects Survival Guide, John Wiley & Sons, New York, 1996.

Otero, S., A Real-time Distributed Analysis Automation for Hurricane Surface Wind Observations , Master Thesis, School of Computer Sciences, Florida International University, Miami, FL, in progress.

Schach, S. R., *Classical and Object Oriented Software Engineering, Third Edition*, Irwin, Boston, 1996.

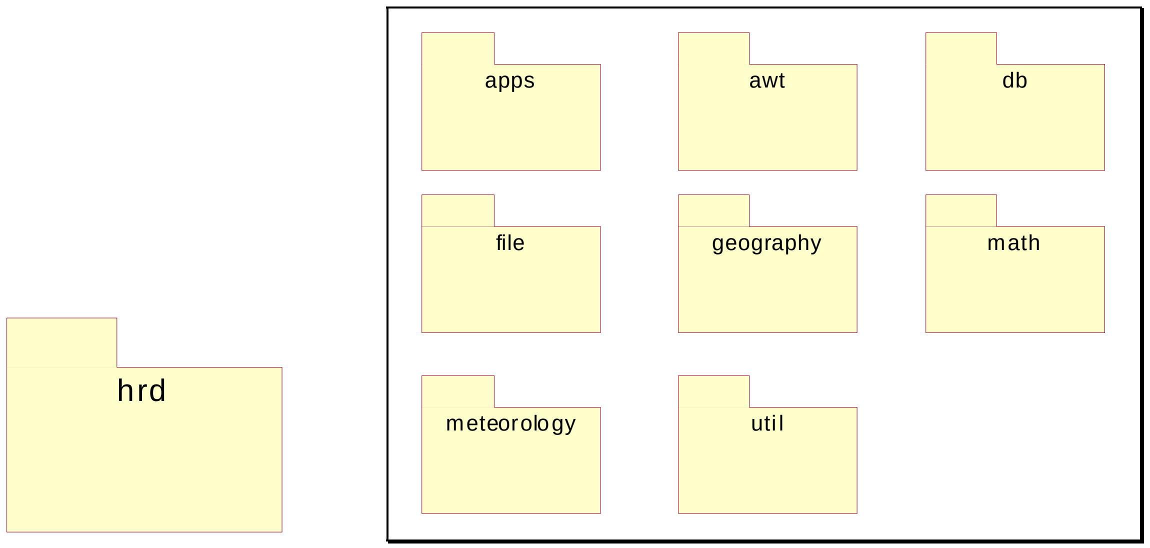
Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., Lorensen, W., *Object Oriented Modeling and Design*, Prentice Hall, Englewood Cliffs, NJ, 1991.

Yourdon, E., *Object-Oriented Systems Design: An Integrated Approach*, Prentice-Hall, Englewood Cliffs, NJ, 1994.

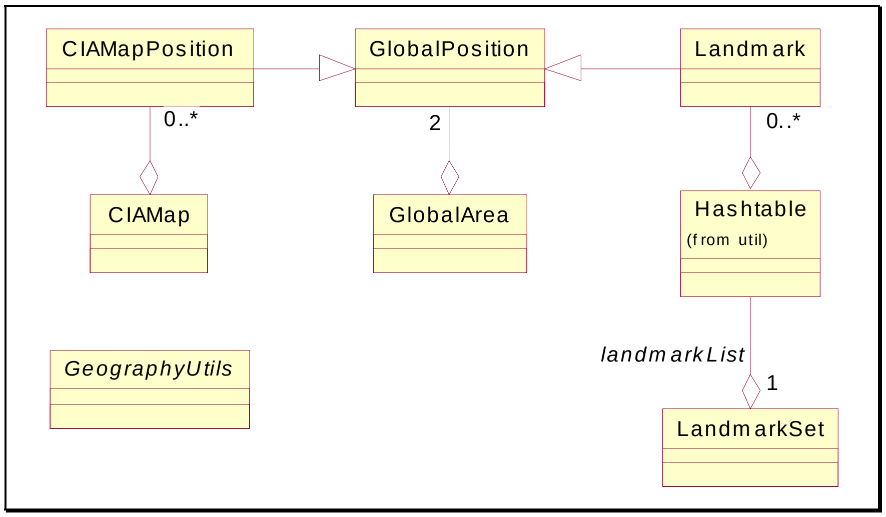
Appendix: Class Hierarchies

Below I present the class diagrams for the classes in the HRD packages. The notation follows that of the UML object oriented design model. For purposes of clarity, these diagrams do not include attributes and operations. Instead they only demonstrate relationships between classes and interfaces.

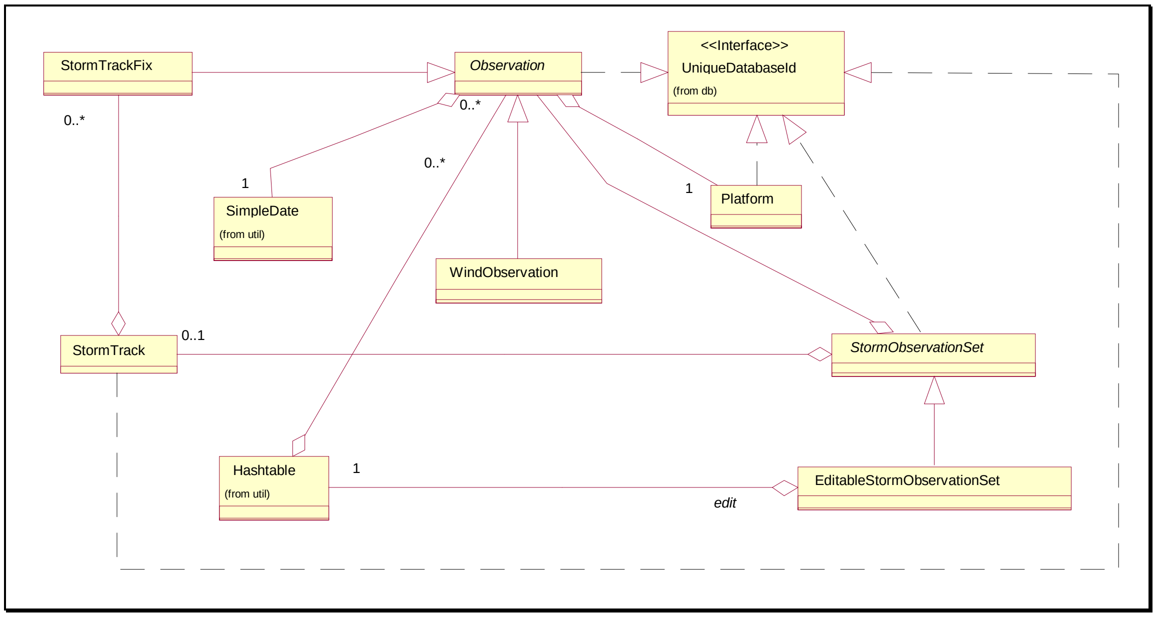
- The HRD package hierarchy. The "hrd" package is the root of the hierarchy. All packages on the right are contained within the "hrd" package. Other packages are also contained within some subpackages.



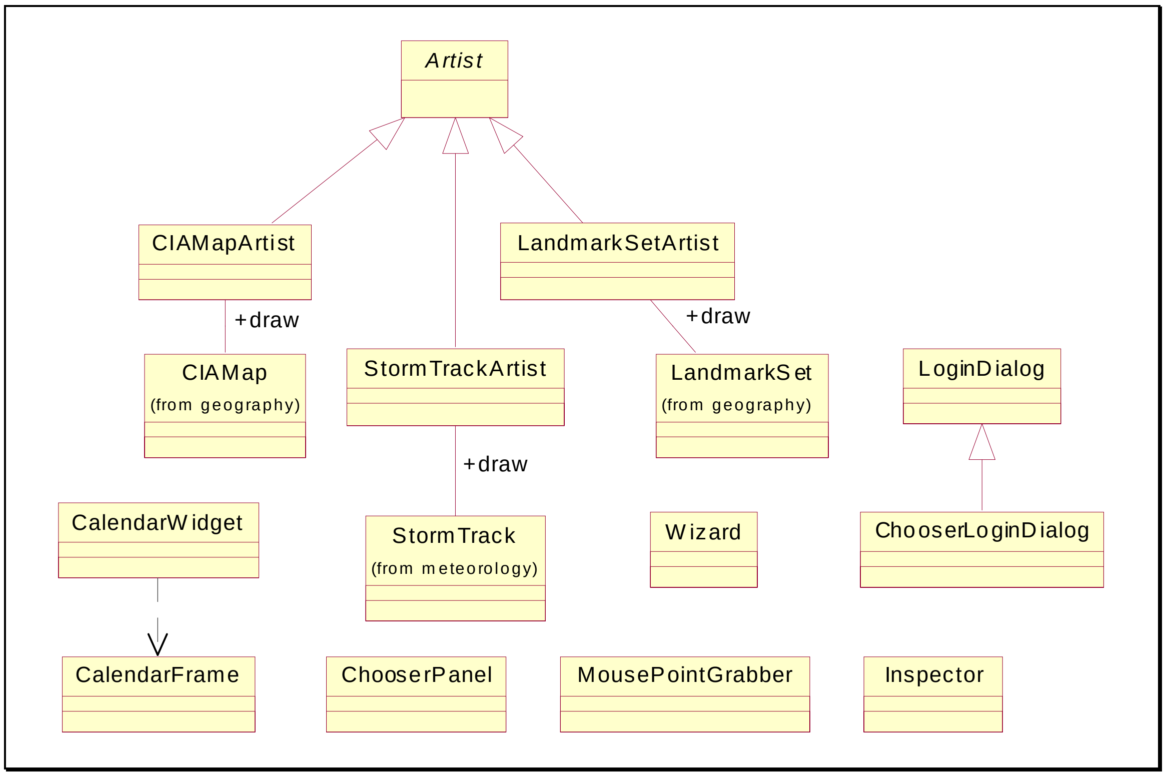
- The *hrd.geography* package contains all classes which deal with coordinates on the globe, geographic areas and mapping.



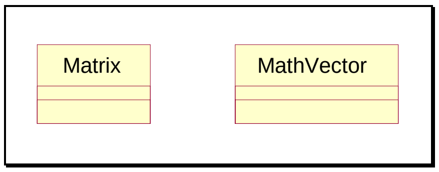
- The *hrd.meteorology* package contains classes pertaining to meteorology, data collection and observations in general.



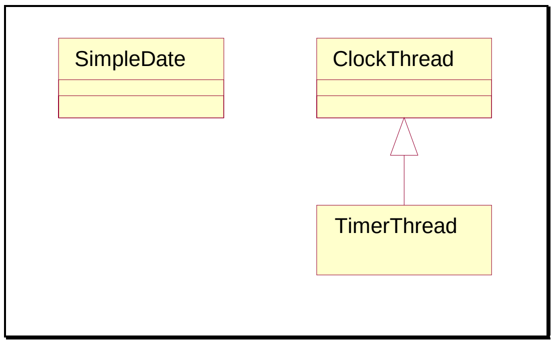
- The *hrd.awt* package contains specialized graphical user interface base level components such as wizards and inspectors as well as abstract and concrete classes for drawing plot components.



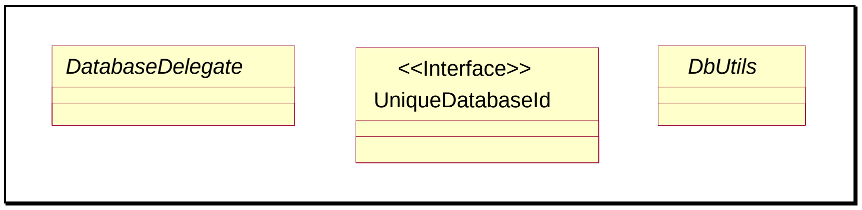
- The *hrd.math* package contains the vector and matrix classes currently used for graphical object transformations such as rotation scaling and translation.



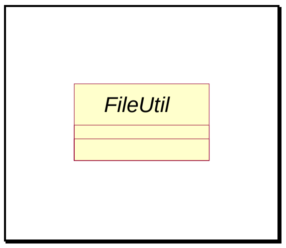
- The *hrd.util* package contains some non-graphical utility classes used to help simplify implementation.



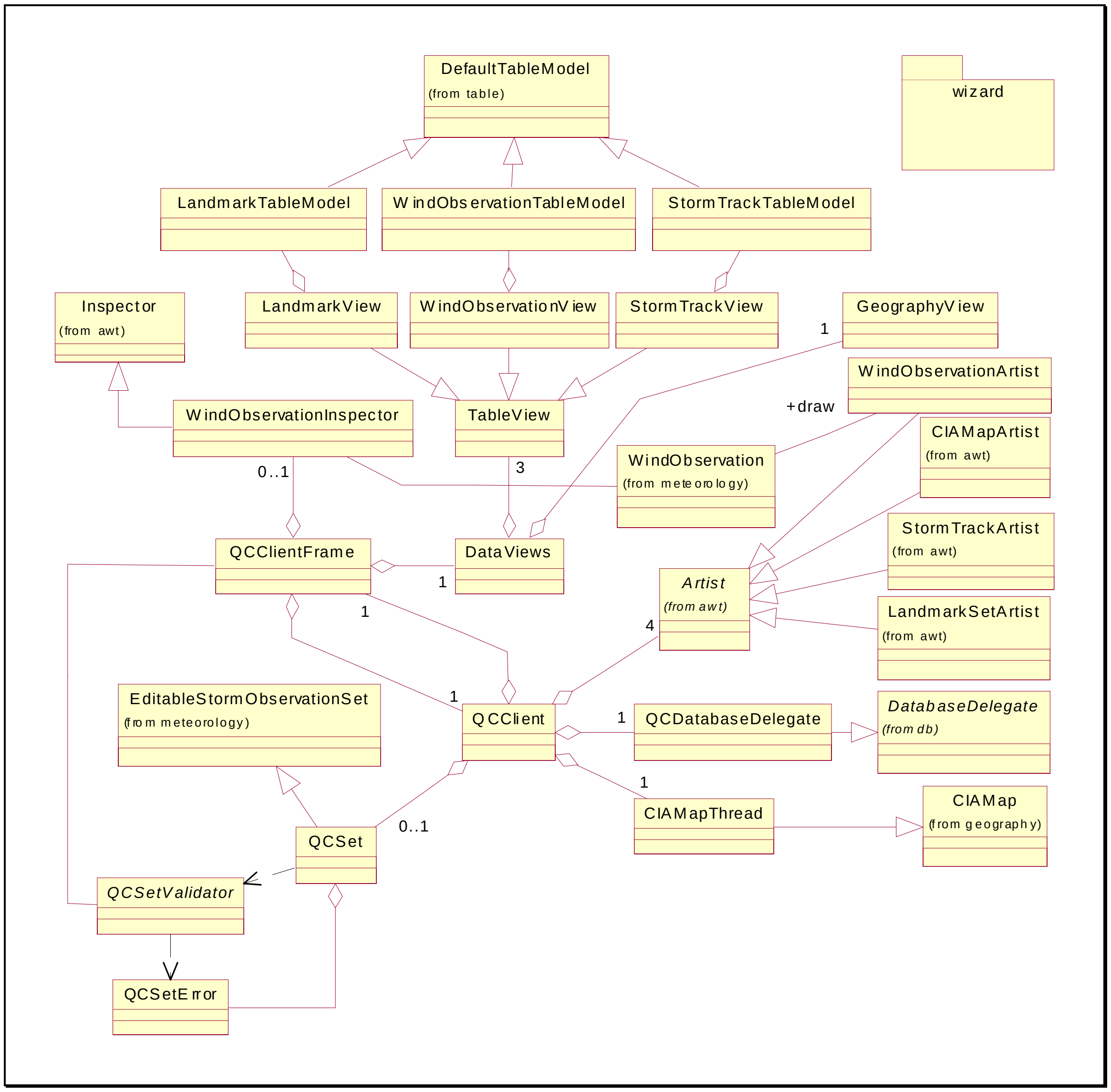
- The *hrd.db* package contains interfaces and abstract classes that are used by other classes to help provide a more customized and integrated, yet flexible JDBC database programming layer for a variety of applications.



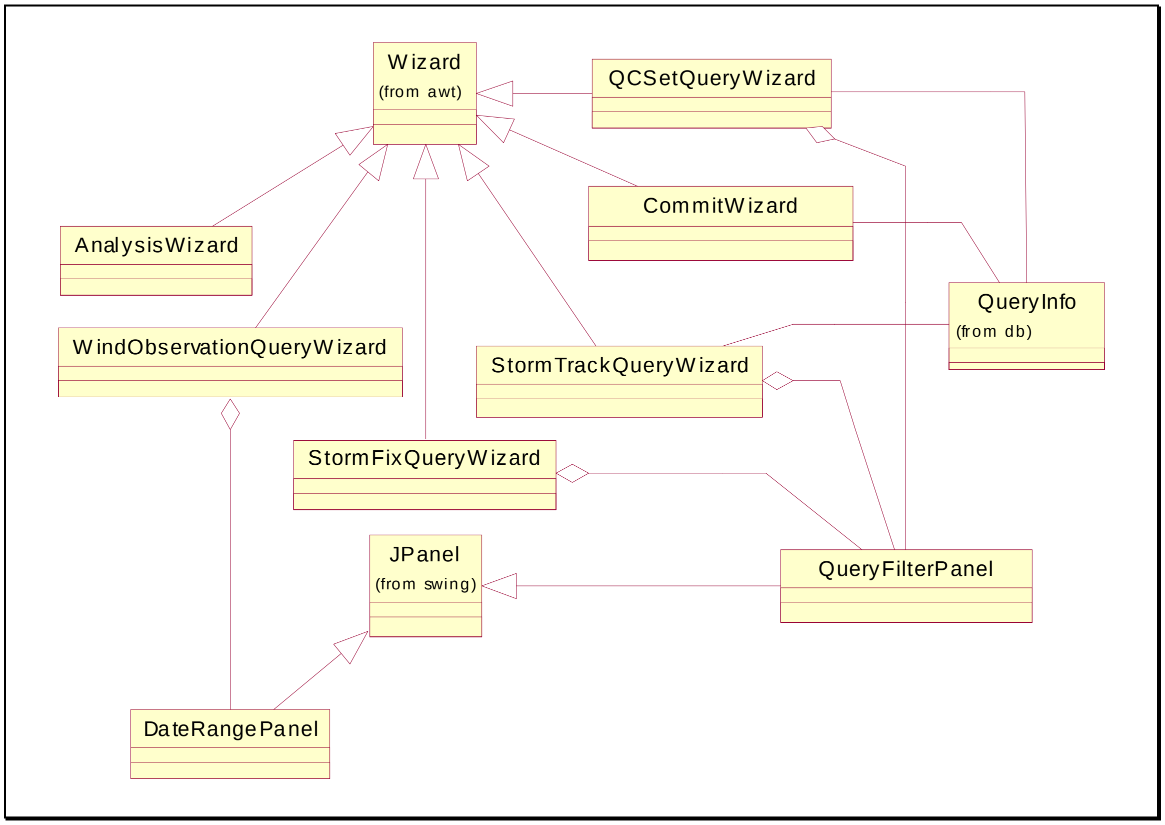
- The *hrd.file* package currently contains a single class of static methods that implement abstract filesystem operations.



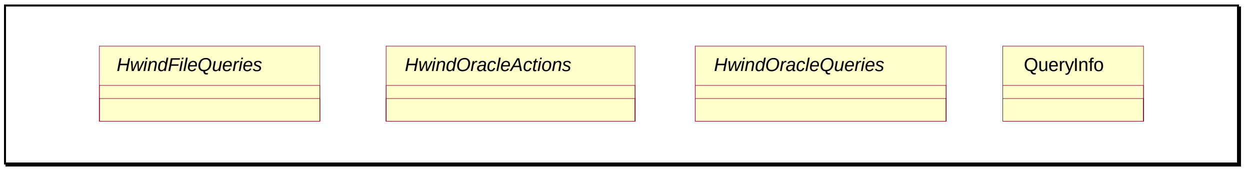
- The main application classes are contained within the *hrd.apps.hwind.qcclient* package. Here, higher level classes are subclassed, if necessary, to a degree that they provide functionality specific to the immediate problem addressed by the application. For example, QCSet is derived from *hrd.meteorology.EditableObservationSet* which is, in turn, a child of the *hrd.meteorology.StormObservationSet* class. A QCSet provides the QCClient with specific attributes and operations that it requires.



- The *hrd.apps.hwind.qcclient.wizard* package contains all of the implementations of the specialized wizard interfaces that are used by the QCClient application. These wizards are used primarily to help the user select conditions for database and filesystem queries and actions.



- The *hrd.apps.hwind.db* package contains abstract classes with any specialized JDBC code required by the H*WIND application. All database specific queries and actions should be hidden within the static methods in those classes.


HwindFileQueries
HwindOracleActions
HwindOracleQueries
QueryInfo